Hindawi
International Journal of Reconfigurable Computing
Volume 2018, Article ID 3276159, 25 pages
https://doi.org/10.1155/2018/3276159

Research Article

Modelling and Assertion-Based Verification of Run-Time Reconfigurable Designs Using Functional Programming Abstractions

Bahram N. Uchevler and Kjetil Svarstad

Department of Electronics and Telecommunication, Norwegian University of Science and Technology, Norway

Correspondence should be addressed to Bahram N. Uchevler; najafiuc@iet.ntnu.no

Received 7 February 2018; Revised 14 May 2018; Accepted 13 June 2018; Published 10 July 2018

Academic Editor: João Cardoso

With the increasing design and production costs and long time-to-market for Application Specific Integrated Circuits (ASICs), implementing digital circuits on reconfigurable hardware is becoming a more common practice. A reconfigurable hardware combines the flexibility of the software domain with the high performance of the hardware domain and provides a flexible life cycle management for the product with a lower cost. A complete design and assertion-based verification flow for Run-Time Reconfigurable (RTR) designs using functional programming abstractions of Haskell are proposed in this article, in which partially reconfigurable hardware is used as the implementation platform. The proposed flow includes modelling of RTR designs in high levels of abstraction by using *higher-order functions* and *polymorphism* in Haskell, as well as their implementation on partially reconfigurable Field Programmable Gate Arrays (FPGAs). Assertion-based verification (ABV) is used as the verification approach which is integrated in the early stages of the design flow. Assertions can be used to verify specifications of designs in different verification methods such as simulation-based and formal verification. A partitioning algorithm is proposed for clustering the assertion-checker circuits to implement the verification circuits in a limited reconfigurable area in the target FPGA. The proposed flow is evaluated by using example designs on a Zynq FPGA as the hardware/software implementation platform.

1. Introduction

The exponential scaling of feature size has led to an exponentially growing number of transistors in integrated circuits. In order to improve the productivity of the designers with the continuously growing complexity of circuits, a great deal of innovation is required in the development of high-level design and verification methodologies.

Describing designs in high levels of abstraction enables the designer to do a broad design space exploration (DSE) to achieve better solutions. VHDL and Verilog are the two most common hardware description languages (HDL) used for describing synthesizable digital circuits. In order to raise the abstraction-level of design descriptions, an HDL based on a functional programming language such as Haskell can provide new design abstractions with higher-order functions and polymorphism [1]. Higher-order functions and polymorphism of Haskell hide the unnecessary details and increase the productivity of the designer. With the increasing nonrecurring engineering costs and long time-to-market for Application Specific Integrated Circuits (ASICs), implementing digital circuits on reconfigurable platforms such as Field Programmable Gate Arrays (FPGAs) is becoming a more common practice. Reconfigurable hardware offers a flexible life cycle management of software as well as the high performance of hardware at the same time. Some of the reconfigurable platforms such as the recent FPGAs from Xilinx support partial reconfiguration which enables reconfiguration of a user-selected area in the programmable logic on the fly while the other parts of the circuit are functioning without interruption. This is also referred to as Dynamic Partial Reconfiguration (DPR) or partial reconfiguration (PR) for short [2].

Using the high-level abstractions of Haskell to model Run-Time Reconfigurable (RTR) designs to be implemented on partially reconfigurable hardware enables the designer

to perform DSE in the early stages of the design flow and increases the productivity of the designer by abstracting away the unnecessary details of the design and the flow. Functional programming abstractions can also play an important role in the verification of digital designs. Verification takes more than 50 percent of the whole development time and effort in most of today's complex designs [3]. Functional HDLs provide a mathematical description for functions and are more suited for formal verification compared to other high-level imperative languages such as C.

Describing verification circuitry together with the Design Under Verification (DUV) in high levels of abstraction enables the designer to use a *unified* language for design and verification, which offers an easier development and integration of verification expressions with the DUV by using parametrization, higher-order functions, polymorphism, and other high-level structures and functions of Haskell.

One of the recent techniques for accelerating the verification process is to implement the verification functions on hardware together with the DUV [4–6].

In some complex SoC designs, it is common to have hundreds or thousands of assertions checking different behaviours of the design [7, 8]. Placing all the assertion checkers of the design in one verification module will lead to a large, resource demanding module, and the amount of the available resources on the target implementation platform might not be enough to implement the verification module. Thus, implementing such a resource demanding module on hardware can be challenging for the designer. Implementing the verification circuitry on partially reconfigurable platforms enables swapping different verification circuits in the partially reconfigurable area on the fly, which increases the flexibility of the verification process. Haskell is used to describe RTR designs and synthesizable verification functions in high levels of abstraction with a unified language in this work, which leads to an integrated design and verification flow targeting partial reconfigurable platforms.

CAES Language for Synchronous Hardware (CλaSH) is a Haskell-based HDL which is used as the design description language in this work. It is also the name of the tool that is used for simulating Haskell descriptions and converting into synthesizable VHDL/Verilog code. CλaSH has a distinct advantage over other functional languages such as Lava and ForSyDe in terms of user-defined data types, being the only language that supports the full range of choice constructs, especially pattern matching. A more detail comparison of CλaSH with other functional HDLs can be found in [1].

The main contributions of this article are summarized as follows:

(i) Modelling of RTR designs with CλaSH (Haskell), in higher levels of abstraction by using *higher-order functions* and *polymorphism* of Haskell.

(ii) Proposing and implementing a new flow for implementing RTR designs on FPGAs with PR support, by integrating the latest PR design flow from Xilinx, and automatic generation of simulation-only VHDL description of the RTR design, partial and full bitstreams and a reconfiguration management software.

(iii) Design and implementation of synthesizable basic building blocks for assertion checkers in Haskell.

(iv) Expressing Boolean layer and temporal layer properties of designs in Haskell and their automatic composition from the basic building blocks.

(v) Using a unified description language, Haskell, for describing the DUV, its verification expressions (assertions), and RTR structures.

(vi) Proposing a complete flow for modelling, assertion-based verification, and implementation of RTR designs.

(vii) A partitioning algorithm for mapping verification circuitry to partially reconfigurable regions in FPGAs.

Section 2 provides the background for the rest of the paper. Section 3 explains the proposed high-level modelling of RTR designs with CλaSH. The proposed modelling approach is generic and is not limited to a specific domain. Section 4 briefly presents the state of the art for implementing assertion checkers on hardware, and Section 5 explains how hardware assertion checkers are implemented with CλaSH. An RTR implementation of the proposed assertion checkers is discussed in Section 6 as an specific application of the proposed modelling approach. A unified flow for design and verification is presented in Section 7 and the paper is concluded in Section 8.

2. Background

2.1. Partially Reconfigurable Hardware. Implementing computationally expensive parts of a design as ASIC can lead to nonflexible, high performance, and low-power solutions and cost a lot of time and money. In contrast, software implementation of them may result in a lower development time and lower performance. Reconfigurable hardware fills the gap between ASIC and software solutions by combining the flexibility of software and the high performance of ASIC, which makes it an appropriate option when both flexibility and performance are required at the same time.

The need for reconfigurability can be driven by 3 main factors: *Multiability* (to perform different functions at different times), *Evolvability* (to adapt to the environment and changes in standards over time), and *Survivability* (the system remains functional despite having a few failures) [10]. Different architectures have been proposed for reconfigurable hardware, but FPGAs are the most common reconfigurable hardware in practice. Some FPGAs provide partial reconfiguration at run-time, which means it is possible to change some parts of the circuit on FPGA, without affecting the operation of the rest of the circuit functioning on the FPGA fabric. This feature is called DPR or PR for short. There are a few design flows available from Xilinx on implementing RTR designs on FPGAs with PR support. The PR supported in Xilinx's Vivado is a script-based flow and easier to integrate with custom design flows compared to the old PR flows. The Vivado-based PR flow is supported by the 7-series FPGA families such as Kintex7, Zynq7000, and Virtex7 [2]. As shown in Figure 1, the designer can specify reconfigurable regions (RRs) in the

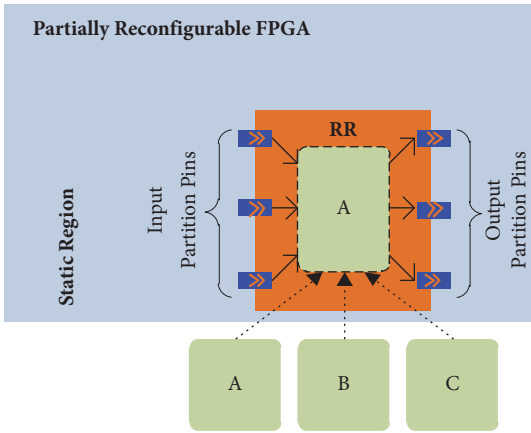


Figure 1: An RTR system on Xilinx FPGAs.

FPGA fabric and change the functionality of the region at run-time by downloading the partial bitstream of the desired module, as long as it has the same communication interface with the static part of the design. Each RR needs specific communication structures bound to a fixed location in the FPGA tiles to communicate with the static part. These communication structures are usually called *bus-macros* or *proxy logic* made of Look-Up-Tables (LUTs) mapped to specific tiles in FPGA fabric array by using physical constraints in the design tools [2]. The alternative candidates for the specified RR are provided by the user (`A`, `B`, and `C` modules in Figure 1) which have the same interface and are called reconfiguration candidates (RCs) or reconfigurable modules (RMs) of the RR. Only one of these reconfigurable modules can be configured in the RR at once, and this module is called the *active* reconfigurable module. The generated partial bitstreams can be programmed through Xilinx's standard programmer or through Internal Configuration Access Port (ICAP) by the embedded processor on the target FPGA. Therefore, the embedded processor can be used for running reconfiguration management software along with other applications.

2.2. Haskell for Hardware Design. HDLs such as VHDL and Verilog are common and conventional for describing detailed hardware, but they can be cumbersome for describing higher levels of abstraction such as polymorphism, higher-order functions, and parametrization in larger designs [1].

In this work, we use Haskell as the hardware description language. Haskell is a functional programming language in which computation is similar to the evaluation of mathematical functions [1]. Being close to mathematical and formal description of hardware enables the designer to avoid exhaustive tests of large designs. This features make the designs described by Haskell more amenable to formal verification. Formal verification is also possible for imperative languages; however, they are not self-describing in the way functional languages are. Furthermore, the program transformation is also easier for functional languages compared to imperative languages [11].

Modelling languages such as SystemC are also used for describing hardware designs in higher levels of abstraction,

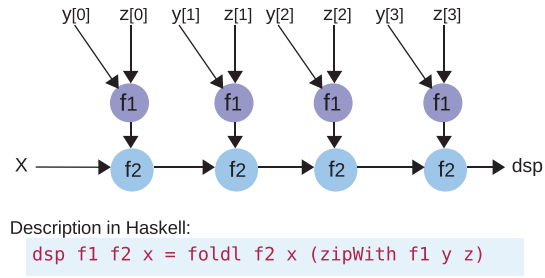


Figure 2: Description of a parametric module in Haskell.

but most parts of the high-level structures are not synthesizable and are mainly for simulation purposes. We use CλaSH to translate designs described in Haskell into synthesizable VHDL code. CλaSH is an experimental tool that accepts a subset of Haskell as its input language and provides libraries for modelling hardware in Haskell [1]. The following features offer the possibility of exploiting higher levels of abstraction and generality for hardware description:

(i) *Higher-order functions*: functions in functional HDLs can receive other functions as their input arguments, or return functions as their return value. This feature is used for parametrizing different parts of the design.

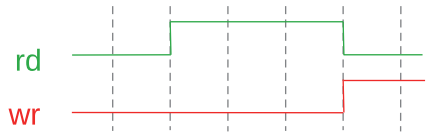
(ii) *Polymorphism*: functions in functional HDLs are not tied to a specific type. Types are inferred for the functions and the designer can use polymorphism to apply a function on variety of data types.

Figure 2 shows an example circuit described in Haskell in which parametrization and higher-order functions are used to shape a general circuit. `zipWith` function generates a vector of elements with applying a function on elements of its two input vectors. Each element in the output vector is the result of applying a function `(f1)` on corresponding elements from the input vectors (`y` and `z`). Another useful function used in this example is `foldl`. It *folds* a vector of elements with a function `(f2)`, starting from left, with an initial value `X`. In this example, functions `f1` and `f2` and the initial value `X` are passed as parameters to the `dsp` function. It is possible to specialize the structure to a digital filter just by applying `multiplication` and `addition` functions to `f1` and `f2`, respectively (i.e., by calling `dsp (*) (+) 0`).

2.3. Design Properties and Assertions. Design properties are Boolean expressions which express the behaviour of a design. Assertions are set on design properties, and any violation from a property is reported as an error. The Accellera Property Specification Language (PSL) and System Verilog Assertions (SVA) are the two common languages for property description. The main part of these languages is based on the Linear Temporal Logic (LTL), augmented with regular expressions to overcome the limitations in the expressiveness of LTL [12]. These property description languages can be used at different levels of abstraction during the design process from high, transaction-level down to low gate-level implementation of the design. In the verification stage, assertions are used to detect violations from the described

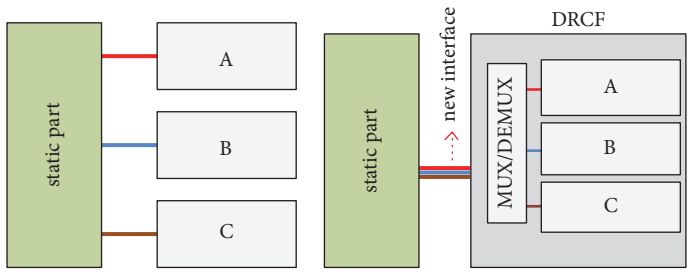
```
// A verification expression with PSL sequences
{ rd[*3] } |=> { wr }

// A verification expression with Boolean expressions
rd && next(rd) && next[2](rd) -> next[3](wr)
```

(a)



(b)

Figure 3: (a) A PSL sequence and the equivalent PSL Boolean expression. (b) A matching diagram for the expression.



(a) (b)

Figure 4: (a) The original design. (b) Grouping RMs to a DRCF.

properties which can be done through *dynamic verification* (e.g., simulation) or *static verification* (e.g., theorem provers for formal verification). In addition, it provides standard means for designers to describe formal specification of the design for documentation purposes. Usually a property is built from *operators* and *signals* of the design. An example assertion on a design property in PSL is expressed as follows:

`always rose(ready&&grant)-> next [2](wr).`

In this example, a master is expected to write its data into the bus exactly two clock cycles after it has received the *grant* and *ready* signals from the arbiter and the slave respectively; otherwise, an assertion will trigger to indicate an error.

In addition to Boolean expressions, there are Sequential Extended Regular Expressions (SEREs) in PSL which describe the multicycle behaviour of the DUV. The simplest form of an SERE is a set of Boolean expressions separated by semicolons such as `{a;b;c}` on three inputs `a`, `b`, and `c`. This SERE describes a scenario in which the input signal `a` is *true* in the first clock cycle, `b` is *true* in the second clock cycle, and finally `c` is *true* in the third clock cycle. Every semicolon represents a clock cycle border in the behaviour of the sequence. Generally, SEREs enable the designer to describe a sequence of Boolean expressions in consecutive cycles. Additional operators such as *repetition* operators are used to represent a compact description of some scenarios. For example, the operators `[*N]` and `[*]` are used on SEREs to represent a compact description of the activation of a signal for *exactly* `N` consecutive clock cycles, or *any* number of clock cycles, respectively, starting from the current clock cycle. The sequence implication operator `|=>` is a nonoverlapping implication which means, in the expression `S1|=>S2`, the first cycle of sequence `S2` begins after the last cycle of `S1` ends. In contrast, the overlapping implication operator `|->` in the expression `S1|->S2` assumes the first cycle of `S2` has an overlap with the last cycle of `S1`.

A specific behaviour of a design can be expressed in different ways using Boolean expressions or SEREs in PSL. For example, the design behaviour shown in Figure 3(b) is described using two different expressions shown in Figure 3(a). More details about operators of PSL are presented in [12]. Property description languages such as PSL or SVA are usually used for describing design properties and assertions which are simulated using simulation tools such as ModelSim alongside the DUV. Depending on the size and complexity of the DUV, hundreds or thousands of assertions are described, each checking for a specific behaviour of the DUV [8].

3. High-Level Modelling of Run-Time Reconfigurable Designs

Modelling of dynamic reconfigurable systems is done with SystemC in [13] which is one of the most referred works in this topic. The main contribution of that work is to collect all RMs of an RR into a module called Dynamically Reconfigurable Fabric (DRCF) and configure a module to the RR when it is requested from other parts of the design. Since the interfaces with the static part can differ for each RM, the containing DRCF module must provide a superset of interfaces of all the RMs for a specific RR. This approach provides a DSE in higher levels of abstraction by assigning the existing modules of a design into different RRs without being involved in the details of the partial reconfiguration process.

Figure 4 shows an example with three reconfigurable modules before and after using a DRCF component. The RMs are embedded in the DRCF component and accessed by their corresponding interfaces and a multiplexer/demultiplexer in the DRCF component. The whole process of including the DRCF component in the design is divided into the following steps. First an analysis of the existing modules is done to extract their interfaces, and all the instances of the modules

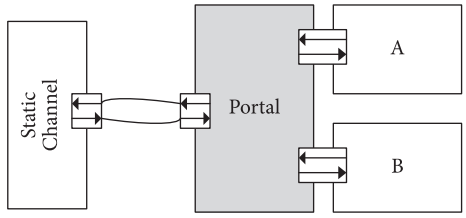


Figure 5: A portal connecting dynamic and static parts, adapted from [9].

targeting the same RR are located and analysed at the same level of hierarchy. After this phase, a DRCF component is created and instantiated from a template which includes all the interfaces and ports collected in the analysis phase. The DRCF module should contain all the reconfigurable alternatives for an RR in the same level of hierarchy. When a call to a specific RM is issued, it becomes active in the DRCF, if it is not already the active module in the DRCF component.

The modelling approach presented in [9], which is also using SystemC as the high-level modelling language, is based on the concept that reconfiguration of RMs on a specific area on the target hardware can be modelled by changing the connections between RMs and the static part. This reorganizing the communication between static and dynamic parts is achieved by using Reconfigurable Channels (ReChannel) named *portals*. A portal is a switch that is designed to connect a static channel to ports of the RMs. As shown in Figure 5, several RMs can share a portal to connect to the static part and only one of the them can be connected to the static part at a time. The connection between static and reconfigurable parts of the design is established by forwarding the events and calls from the static part to RMs by means of `event_forwarder` and `Accessor` components, respectively.

In [14, 15] authors use UML and design patterns to explore the design space in a multiprocessor system on programmable chip to capture dynamic properties of both the application and the architecture. Standard UML elements and some well known design patterns are used to model RTR designs. RMs are specified by means of design patterns. Two well known design patterns *strategy* and *state*, which are traditionally used in software domain, are used for this purpose. These design patterns are used to select alternative algorithms at run-time in an application. There are similar works on modelling RTR designs in high levels of abstraction using high-level languages such as SystemC in [16–18] with more focus on simulation or using UML in [19–22] for managing components of the RTR design described mainly with VHDL/Verilog.

3.1. Describing RTR Designs with CλaSH. The high-level modelling approaches mentioned so far, either do not support implementing RTR designs on hardware (such as SystemC based approaches) or are more suitable for integrating already existing designs rather than designing custom FPGA-based systems (such as UML based approaches). This section briefly describes how RTR designs are modelled and implemented on partially reconfigurable FPGAs.

In order to describe an RTR design in high levels of abstraction without being involved in the low-level details of the partial reconfiguration process of the target hardware, the higher-order functions of Haskell are used. Higher-order functions in Haskell provide the ability of passing functions as arguments to other functions. In higher levels of abstraction, an RR can be seen as a programmable *function* with the following features:

(i) Maximum number of input and output ports of the programmed function is limited to the number of input and output ports of the RR, respectively.

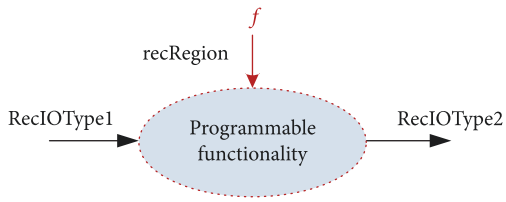
(ii) The maximum amount of hardware resources needed for implementation of the programmed function is limited to the amount of available resources in the RR.

Figure 6(a) shows the representation of a reconfigurable region (`recRegion`) in a high level of abstraction, in which the input-type, output-type, and the programmed function of the RR are represented with `RecIOType1`, `RecIOType2`, and `f`, respectively. Any function with the above-mentioned features can be programmed to the RR. Using the higher-order functions of Haskell, an RR can be represented with a function that receives another function as an input and applies it on its data input, as shown in Figure 6(b). In other words, the function `f` is a placeholder for any function that is compliant with it in input-output types. Using the higher-order functions of Haskell enables the designer to determine and pass the functionality of the reconfigurable region from higher levels of hierarchy from the top level of the design, without applying any change in the lower levels of hierarchy in the design. This is compliant with the hardware implementation of an RTR design on a hardware with DPR support, in which the RR can be programmed with different partial bitstreams from software, without the need to apply changes in the implemented static part of the design.

As shown in Figure 6(b), the reconfigurable region is represented with the higher-order function `recRegion`, which simply applies the received function on input `i` of type `RecIOType1` to produce an output of type `RecIOType2`. The function argument is exported as a port in the hierarchy levels up to the top module, to make the design reconfigurable from the testbench without changing its internal modules.

Figure 7 shows the architecture of the target platform used for implementing RTR designs in this work, together with three reconfiguration candidates `RC1`, `RC2`, and `RC3`, which have the same input and output types. The embedded processor of the target FPGA is used for accessing the reconfigurable fabric, as well as for running the reconfiguration management software. Specific types are defined to be used as input/output types of the RMs and the RR which is stored in a library. Listing 1 shows several of the defined data types with different sizes.

For example, `Signal(BitVector 32)` defines a 32-bit port on the RR and its RMs. The set of reconfiguration candidates is represented by a *vector* of functions with the same type signature. For example, the reconfiguration candidates of Figure 7 are represented with a vector of functions called `rclist` as follows:



(a)

```
recRegion :: (RecIOType1 -> RecIOType2)
                          ->
             RecIOType1 -> RecIOType2

recRegion   f   i  =  f  i
```

(b)

FIGURE 6: (a) High-level abstraction of a reconfigurable region. (b) Modelling a reconfigurable region with CλaSH.

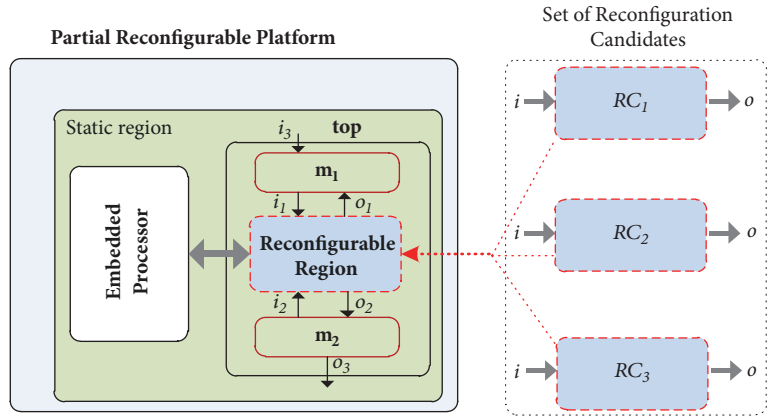


FIGURE 7: An RTR design with three RMs.

`rclist = (rc1:> rc2:> rc3:> Nil)`

which has the following type signature:

`rclist:: Vec 3 (Signal i -> Signal o).`

The `:>` operator is used to attach an element to the vector, and `Nil` represents an empty vector. The vector elements are accessed by their indexes using the indexing operator `!!`. For example, expressions `"rclist !! 0"` and `"rclist !! 2"` return `rc1` and `rc3` functions, respectively. The input/output types `i` and `o` are defined in terms of the reconfigurable types of Listing 1. The structural description for the example design of Figure 7 as well as its list of reconfiguration candidates is shown in Listing 2. Lines 16-18 show how different reconfiguration candidates are passed to the top module as a part of the stimuli from the test bench, to verify the functionality of the design. The first element in the input tuple (e.g., `(rclist!!0)`) specifies the functionality of the RR. The stimuli vector `stimuliVec` is used with the simulation command to apply the stimuli to the design one element at a time. For example, the following command applies the stimuli vector to the top module and shows 4 samples of the output vector in the command line:

`take 4 $ simulate top stimuliVec.`

The operator `$` is used to replace parentheses in Haskell. More details on simulating designs in CλaSH can be found in [23].

3.2. Proposed Design Flow. The proposed design flow is shown in Figure 8. The whole design, including RMs as well as the static part, is described with CλaSH. The functionality of the design in CλaSH is verified by simulating the design using the simulation commands in the CλaSH tool. The reconfiguration candidates are extracted from the vector of functions used in the test bench `(rclist)`. The whole design is translated to VHDL with the CλaSH tool, to do register Transfer Level (RTL) simulation by using the automatically generated VHDL models of the static part, RMs, reconfigurable regions, and their interfacing logic. The extracted information about the reconfigurable regions and their RMs is used to generate both hardware modules through the VHDL-generation feature of the CλaSH tool, and the software for managing the reconfiguration process which is compiled for the embedded processor of the target FPGA. The last steps in the flow include integration of the proposed flow with the latest partial reconfiguration flow of Xilinx, which is briefly explained in the next section.

3.3. Integration with Conventional Design Tools. The proposed flow is fully automated and implemented by a script called `rclash` which integrates the the CλaSH tool with the conventional design tools such as ModelSim and Vivado for simulation and FPGA implementation, respectively. The steps "Extracting Reconfiguration Candidates", "Integration with Xilinx Partial Reconfiguration Flow", and "Reconfiguration Control Software Generation" are fully implemented in Python, while the rest of the steps in the flow are performed by CλaSH, ModelSim, and Vivado which are integrated in the flow. The `rclash` script is accessible at [24]. In order to run the reconfiguration management software on the embedded processor of the target FPGA, a *template* Vivado project, which contains the embedded processor, is integrated with

```
type RecRegSigIOType8 = Signal ( BitVector 8 )
type RecRegSigIOType16 = Signal ( BitVector 16 )
type RecRegSigIOType32 = Signal ( BitVector 32 )
type RecRegSigIOType64 = Signal ( BitVector 64 )
type RecRegSigIOType65 = Signal ( BitVector 65 )
```

Listing 1: Input/output types for RMs.

```
-- Structural description of top module.
-- Functionality of the reconfigurable region
-- is specified with function 'f'.
top (f, i3) = o3
   where
      -- m1 and m2 are functions representing
      -- the module instances
      i1 = m1 ( i3, o1 )
      (i2, o3 ) = m2 o2
      -- recRegion is a function representing the
      -- reconfigurable region
     (o1, o2 ) = recRegion (f, ( i1, i2 ))
-- A vector specifying all RMs
rclist = (rc1:> rc2:> rc3:> Nil)
-- Stimuli for testing the functionality of top
stimuli0=(rclist!!0,val0) --(rclist!!0) = rc1
stimuli1=(rclist!!0,val1) --(rclist!!0) = rc1
stimuli2=(rclist!!1,val2) --(rclist!!1) = rc2
stimuliVec = [stimuli0, stimuli1, stimuli2]
```

Listing 2: Description of top and its test bench with CλaSH.

the RTR (CλaSH) design. The top level wrapper of the CλaSH design is automatically instantiated in the top level module of a template processor-system design, as shown in Figure 9. The main reconfiguration management software is automatically generated in the flow, which includes addresses of the full and partial bitstreams in the memory as well as the initialization of the device and example reconfiguration steps. The designer can implement the desired reconfiguration scheduling in the generated software. The template project uses ZedBoard as its default target, but it can be adapted to other 7-series FPGA boards by changing the implementation target information and constraints in the synthesis script.

The designer can add constraints to connect the signals of the user-design to internal signals from the AXI-Slave or external pins of the FPGA. The modelling, simulation, and implementation of an example design using the proposed flow are shown in the following section.

3.4. A Run-Time Reconfigurable Coprocessor. The architecture of an example AMBA AHB-based RTR design is shown in Figure 10. There are two AHB masters in the design (`master0` and `master1`) connected to two AHB slaves (`ahbSlaveCP` and `ahbSlaveMem`) with a 64-bit AHB bus. The AHB slaves are a shared memory (`memory`) and an RTR coprocessor (`recCP`). The RTR coprocessor is mapped to a reconfigurable region on the target FPGA with 65 input/output ports. The desired number inputs/outputs can be set on an RR during the design phase. There are three reconfiguration candidates in this design for the reconfigurable region, and the RTR coprocessor can be programmed to function as a 64-bit DES block-cipher, a Finite Impulse Response (FIR) filter, or an Infinite Impulse Response (IIR) filter. The AHB bus, masters, and slaves are all described in a relatively high level of abstraction close to mathematical description with CλaSH which is available in [24]. As an example, Figures 11(a) and 11(b) show the structure and the high-level CλaSH description of the IIR filter, respectively. Function `init` in the CλaSH description of the IIR filter returns the vector by dropping the last item, and function `:>` attaches an element to the start of a vector. `xRegs` and `yRegs` are the registers `(D)` for storing `x` and `y` values, respectively. Figure 11(c) shows the CλaSH description of the reconfigurable coprocessor slave on the AXI bus. Function f which defines the functionality of the reconfigurable region is passed as an argument to the reconfigurable region and applied on its input data according to Figure 6. This function is passed to the top level module from the test bench which provides the capability of changing the functionality of the reconfigurable region without changing the internal modules in lower levels of hierarchy. The simplified structure of the RTR coprocessor is shown

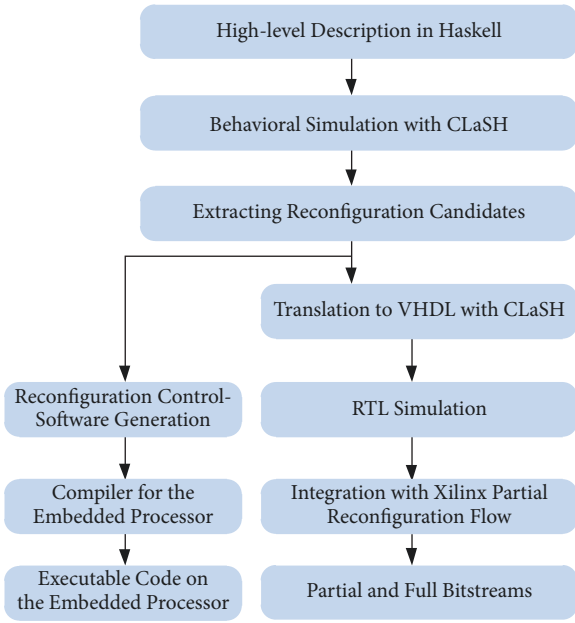


Figure 8: The proposed design flow.

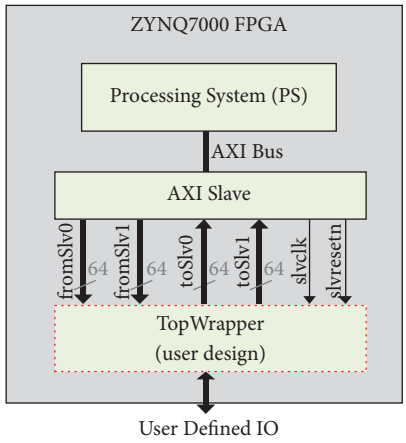


Figure 9: Template processor-based design in Zynq FPGA.

in Figure 12. In the automatically generated simulation-only model of the design, the functionality of the reconfigurable region is simply modelled by multiplexing reconfigurable modules. The generated simulation-only model for the RTR coprocessor is shown in Figure 13.

An *isolation* logic is added at the output of the multiplexer to simulate the delay of the reconfiguration operation as well as sending unknown ('X') values into the static part to simulate the unknown state of the output signals during the reconfiguration process. The simulation result for the reconfigurable coprocessor design using ModelSim is shown in Figure 14, in which the coprocessor is reconfigured from DES to FIR. The `dout0` and `dout1` signals shown in the waveform are connected to the output of the reconfigurable coprocessor on the AHB-slave by default. In addition to the generated full and partial bitstreams, an embedded software for managing the reconfiguration process is also generated by the script and

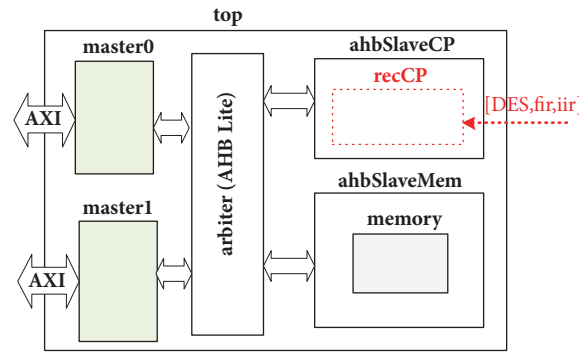


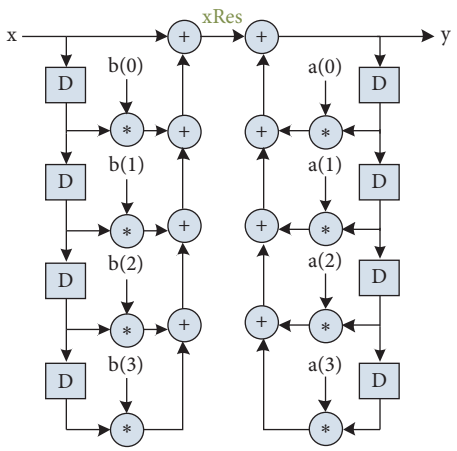
Figure 10: An example RTR design.

copied into the `reconfig` directory in the SDK directory of the generated Vivado project. The generated software which is a C file is called `reconFig. c` can be edited by the designer to apply the desired reconfiguration management on the available partial and full bitstreams. The main part of the autogenerated reconfiguration software is shown in Listing 3. This software assumes that the generated partial and full bitstreams are stored on a SD card attached to the ZedBoard. Each partial bitstream is copied from SD card into a specific address in the DDR memory during the initialization phase in `init_device()` function. The automatically generated addresses are for accessing partial bitstreams on the SD card; however, they can be changed to desired memory addresses by the designer. The function `XDcfg_TransferBitFile` is a Xilinx API used for sending partial and full bitstreams into the FPGA in Xilinx's 7-series FPGAs.

Reconfiguring the reconfigurable region is performed by the function `XDcfg_TransferBitfile`. This is a Xilinx function that provides the access for the embedded processor to write to the configuration memory through the Processor Configuration Access Port (PCAP). The full code of this software is also available in [24]. Currently, only one RR is instantiated and experimented in this work; however, extending the flow to support more than one RR is straightforward.

4. Implementing Assertion Checkers on Hardware

In [3] a modular approach is proposed to generate synthesizable assertion checkers from PSL properties which are composed of logical and temporal operators. The design properties are translated into basic synthesizable components and connections between them are established. The interconnection method is based on the syntax tree of the property. A similar work has been done in [4] on SEREs of PSL which assembles SEREs from synthesizable basic building blocks. Figure 15 shows an example property and its assembled circuit from the basic components. Boulé and Zilic have proposed an automata-based approach for implementing PSL assertion checkers on hardware in [5]. The presented implementation techniques are part of their checker generator tool named MBAC. Figure 16 shows how the MBAC tool is used for assertion-checker circuit generation and integration with the DUV. The DUV, which is described in an HDL, together



(a)

Description in CLaSH:

```
iir (xRegs,yRegs) (a, b,x) = ((xRegs',yRegs'), y)

    where
        xRegs' = x       :> (init xRegs)
        yRegs' = yRegIn  :> (init yRegs)

        yRegIn = foldr (+) xRes (zipWith (*) a yRegs)
        xRes   = foldr (+)  x   (zipWith (*) b xRegs)
        y      = yRegIn
```

(b)

```
ahbSlaveCP (f,(hsel, haddr,hwrite, hwdata, htrans)) = (hready, hrdata)
    where
      (hready ,hrdata, cpdin) = slvIF (hsel, haddr,hwrite, hwdata, htrans,cpdout)
       cpdout                 = recReg f cpdin
```

(c)

Figure 11: Representation of the `IIR` filter (a), (b) and the `Coprocessor` (c) in CλaSH.

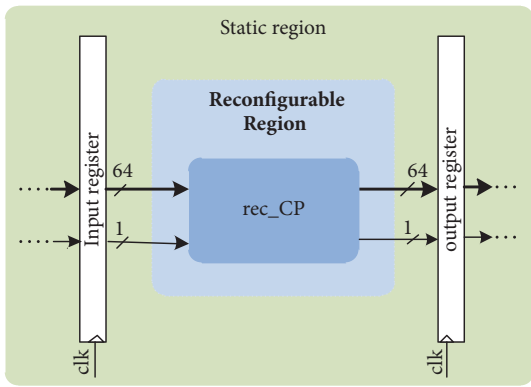


Figure 12: An RTR coprocessor.

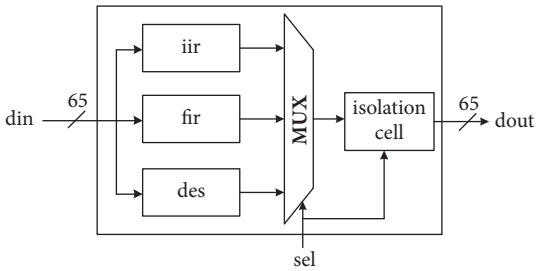


Figure 13: The simulation model of the RTR coprocessor.

with its PSL assertions is inputs to the MBAC tool. MBAC generates synthesizable HDL descriptions of the assertion checkers from the PSL expressions. The generated checker circuits are then integrated with the DUV and implemented together on the target platform. A Nondeterministic Finite Automaton (NFA) is built by combining the base structures which is then transformed into a circuit. The automaton has an initial state, from which the pattern matching begins, and one or more final states. When a final state is active, it means a match is found for the pattern or, in other words, the expression holds. The automaton is translated to a circuit in two steps. First, each state signal is sampled by a D Flip-Flop and its output is referred to as a sampled state signal. Second, a state signal is defined as a disjunction of the edge signals that hit a given state. An edge signal is a conjunction of its symbol and the sampled state signal from which the edge originates. The result signal of the automaton (the signal returned by the automaton) is a disjunction of the state signals of the final states [5].

In the related works mentioned so far, PSL or SVA expressions are translated into synthesizable VHDL or Verilog modules, but integrating assertion checkers in higher levels of abstraction is not addressed. An ABV approach for SystemC designs is presented in [25], which embeds PSL assertions at high levels of abstraction into SystemC code in the form of Abstract State Machines. However, this approach is only for simulation purposes and neither the DUV nor the properties are translated to hardware.

Another work on embedding synthesizable assertion checkers is done in [26]. The authors used Impulse-C as the system-level design language to represent the DUV and its assertion checkers. The described design properties are limited to a specific type of properties and there is no support for multicycle properties such as PSL sequences or LTL-based design properties, which play an important role in ABV.

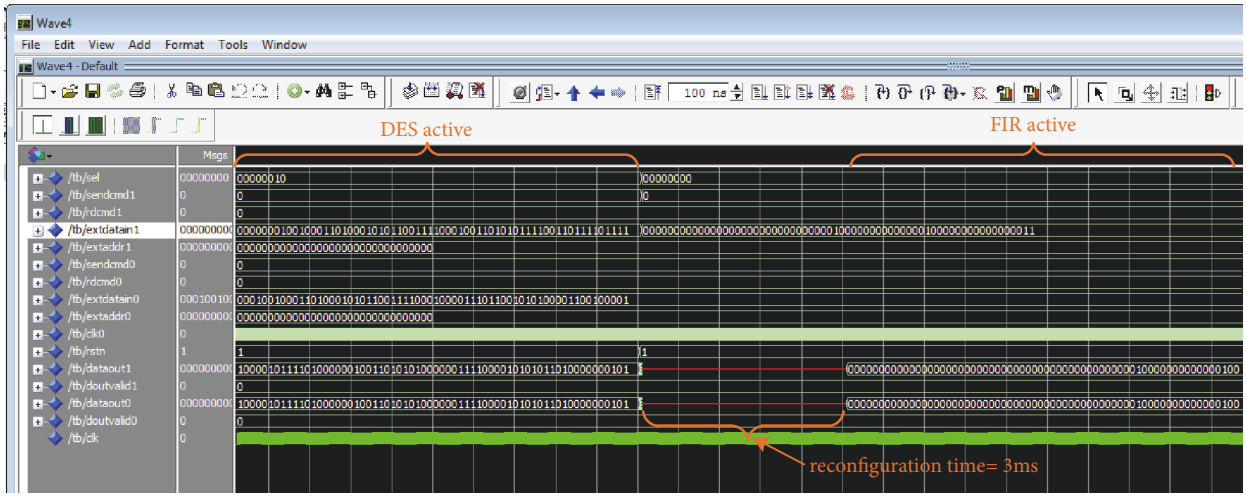
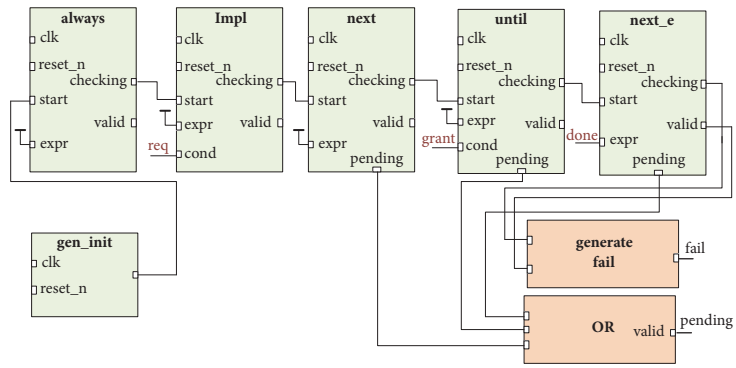


Figure 14: Reconfiguring the region from DES to FIR.




```
property p1 is always req -> (next(next_e 1:2(done)) until grant)
```

Figure 15: An example for modular approach, adapted from [3].

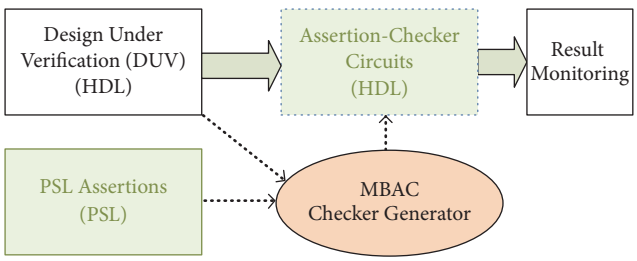


Figure 16: Generating assertion-checker circuits with MBAC, adapted from [5].

Implementation of some of the basic LTL operators using Arrows of Haskell is discussed in [27]. Designing circuits directly with Arrows is complex and limited, since only a few Arrow-composition operators can be used to compose more complex circuits [27].

5. Assertion Checkers in CλaSH

This section explains how Boolean and temporal properties of a design and their assertion checkers are described and integrated in high levels of abstraction with CλaSH, and how they are assembled automatically from basic building blocks.

Design properties and their assertion checkers are composed of Boolean operators (such as `or`, `and`, `not`, and `implication`) or temporal operators (such as `always`, `eventually!`, and SERE operators) and their operands [12].

In this work, each of these operators is defined as a Haskell function in CλaSH. In Haskell, functions can be combined in variety of ways to compose more complex functions. Since each Haskell function in CλaSH is translated into an equivalent synthesizable VHDL entity, design properties and checkers described as compositions of Haskell functions, are translated into synthesizable building blocks connected according to the syntax tree of the expression. As an example consider the following property:

property1: always, if both signals 'a' and 'b' are True, signal 'c' or 'd' must be True.

A possible PSL representation of the property can be as follows:

```
property1 is always ((a and b) -> (c or d)).
```

This property is expressed in CλaSH using the following expression:

```
property1 = always ((a /\ b) --> (c \/ d)).
```

```
1  ////// Define addresses of RMs in the DDR RAM
2  #define DES_addr_in_ddr XPAR_DDR_MEM_BASEADDR+0*0x60000U
3  #define FIR_addr_in_ddr XPAR_DDR_MEM_BASEADDR+1*0x60000U
4  #define IIR_addr_in_ddr XPAR_DDR_MEM_BASEADDR+2*0x60000U
5  int main()
6  {
7  //Initialize the base platform
8    init_platform();
9    init_devices();
10 // User code can be added here to manage the
         reconfiguration. An example code for configuring the
          DES module into the RR is as follows:
11 //      status = XDcfg_TransferBitfile(
12 //                          XDcfg_0,
13 //                          DES_addr_in_ddr,
14 //                          (BITFILE_LEN / 4));
15 // Check if the reconfiguration was successful
16 // if (status != XST_SUCCESS) {
17 //     xil_printf("error!"); return XST_FAILURE; }
18 //     xil_printf("DES is configured to the coprocessor!");
19       return 0;
20 }
```

Listing 3: The Initial reconfiguration management software.

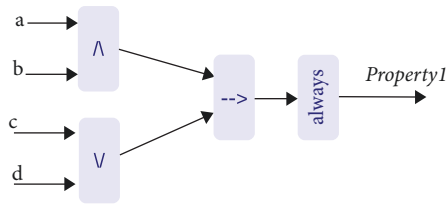


Figure 17: Implementation of `property1`.

Figure 18: Input and output types of an operator building block.

Composition of functions in Haskell is translated by CλaSH into communication ports between building blocks. The building blocks are synthesizable Mealy machines implemented in Haskell themselves. Figure 17 shows the final circuit for `property1` which is formed by connecting the basic building blocks of operators according to the syntax tree of the expression. Blocks `/\`, `\/`, and `-->` are implemented as Haskell functions and represent the Boolean PSL operators `and`, `or`, and `->`, respectively. The Boolean operators implemented in this work are slightly more complicated than the simple logical operators (such as `and, or, or implication`) in which the input and output values can be either `true` or `false`. As an example, consider the following property expressed in PSL:

`p0 is always ( rose(a) -> (a until b) ).`

If the left side of the implication operator does not hold, the property `p0` will hold regardless of the result of the evaluation of the right side. When the left-side holds, the result of the implication becomes dependent on the evaluation of its right side. Evaluation of the expression `(a until b)` in this property can take multiple clock cycles before it is finalized, which means when a rising edge of signal `a` is detected, the implication operator must be able to *wait* until the evaluation of its right side is finished. In addition to the Boolean values (`false` and `true`), each operator can output values to inform the operators ahead, of an ongoing and unfinished evaluation. Since the evaluation result of the right side of the implication might need multiple clock cycles to be finalized, the implication operator must wait until the evaluation of the right side is finished. The inputs and outputs of every implemented operator are of the new 4-value type which is shown in Figure 18. Evaluation of the expressions composing property *p0* is shown in Figure 19.

`True` and `false` values are referred to as *known* values, and the operator input that does not have a known value is referred to as waiting input in this text. As a general rule for the Boolean operators implemented in this work, if an input to the operator is waiting and depends on the signal values in the future cycles, and the result of the operation depends on the value of the waiting input, the evaluation is delayed until the values on inputs are known. However, the other input(s) with the known value will be considered for the evaluation of the operation while waiting for the other input to get a known value. Therefore, the output of the operation can change based on the value of the nonwaiting input, and a specific value called `FalseWeak` is used on the operation output signal to notify other operators consuming its result.

	cycle1	cycle2	cycle3	cycle4	cycle5	cycle6	cycle7	cycle8
a:	False	False	False	True	True	True	True	True
b:	False	False	False	False	False	False	True	True
rose(a):	False	False	False	True	False	False	False	False
(a until b):	False	False	False	Waiting	Waiting	Waiting	True	True
p0:	True	True	True	Waiting	Waiting	Waiting	True	True

time

Figure 19: Evaluation of expressions in property *p0*.

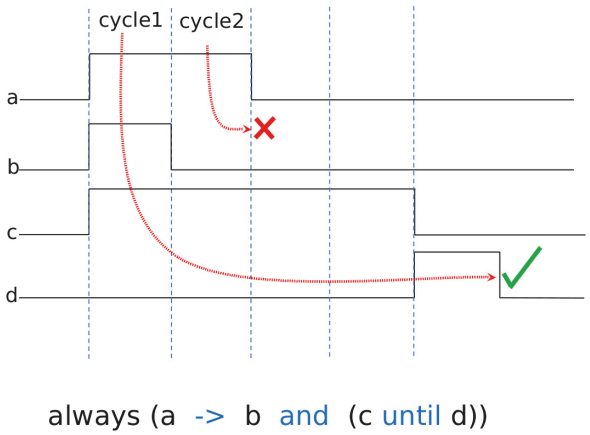


Figure 20: Delaying the evaluation of Boolean operators.

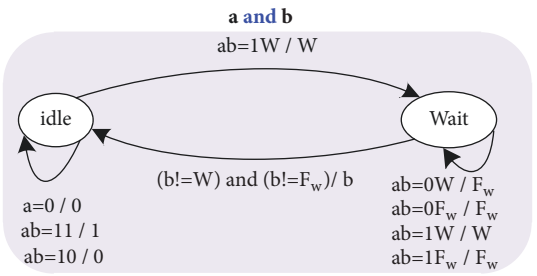


Figure 21: FSM of the 'and' operator.

For example, when an input of the `and` operator is waiting, a `False` value on the nonwaiting input at the current cycle generates the `FalseWeak` value on the output. This is for discriminating between the `False` output value that can be generated as a result of a `False` value on the waiting input in the future and the evaluation result of the operation in the current cycle. As an example consider the property and its evaluation shown in Figure 20 with two subtraces shown starting in `cycle1` and `cycle2`. While the evaluation of the implication operator is delayed because of the `"c until d"` expression, any `False` value on the `b` input of the and operator can evaluate the result of the implication operation to `False` if input `a` is `True` at the same cycle. If `a` was `False` in `cycle2`, the implication operator would still be in the waiting state waiting for a known value and the expression would hold. Considering the multicycle behaviour, the logical operators such as `and` and `or` are not state-less Boolean operations. They are implemented by Finite State Machines (FSMs), since they need to have memory elements to keep track of the past input values. Similarly, the temporal layer operators such as `until` and `before` are also implemented by FSMs. Figure 21 shows the simplified FSM of the `and` operator implemented in this work. For example, while an and operation is waiting on its second input to receive a known value, any `False` value on the first input `( b )` can evaluate the result of the and operation to `FalseWeak`. Values `True`, `False`, `FalseWeak`, and `Waiting` are shown with `1`, `0`, `Fw`, and `W` in the FSMs, respectively.

The property evaluation results `True`, `False`, and `Waiting` correspond to `Holds`, `Fails`, and `Pending` of PSL, respectively. As shown in the FSM of the `and` operator, if an input of the `and` operation is `Waiting`, and the other input is not `False`, the output will be `W` to pass the waiting value to the parent node in the syntax tree of the property. The new 4-value data type is used to represent the input and output values of the LTL operators, instead of the simple two-value Boolean type. In order to keep the complexity of the building blocks low, only a subset of the *simple subset* of PSL is supported, in which only the right side of the implication operator might have LTL operators and the left side is composed of Boolean operators [12], which means the waiting value W can appear only on the right side of the implication operator. The simple subset of PSL is a subset that conforms to the notion of monotonic advancement of time, from left to right through the property. Properties described with simple subset of PSL are more readable. Listing 4 shows two simple properties described in CλaSH for the AHB-based design presented in Section 3. The `sig` function is used to convert a Boolean signal value to the four-value signal introduced earlier in this section. The `neg` function is used for negating its input. The PSL SEREs supported in this work are in the form of `S1 |=> S2`, in which the operator `|=>` is the *suffix implication* operator, and `S1` and `S2` can be sequences of Boolean expressions on the input signals from the DUV. The overlapping implication operator `|->` is not implemented in this work. Any expression composed of the suffix implication operator of the form `S1 |=> S2` is evaluated in the following way in PSL:

The *expression* `S1 |=> S2` *holds if any recognition of the* sequence `S1` *in a signal trace leads to at least one recognition*

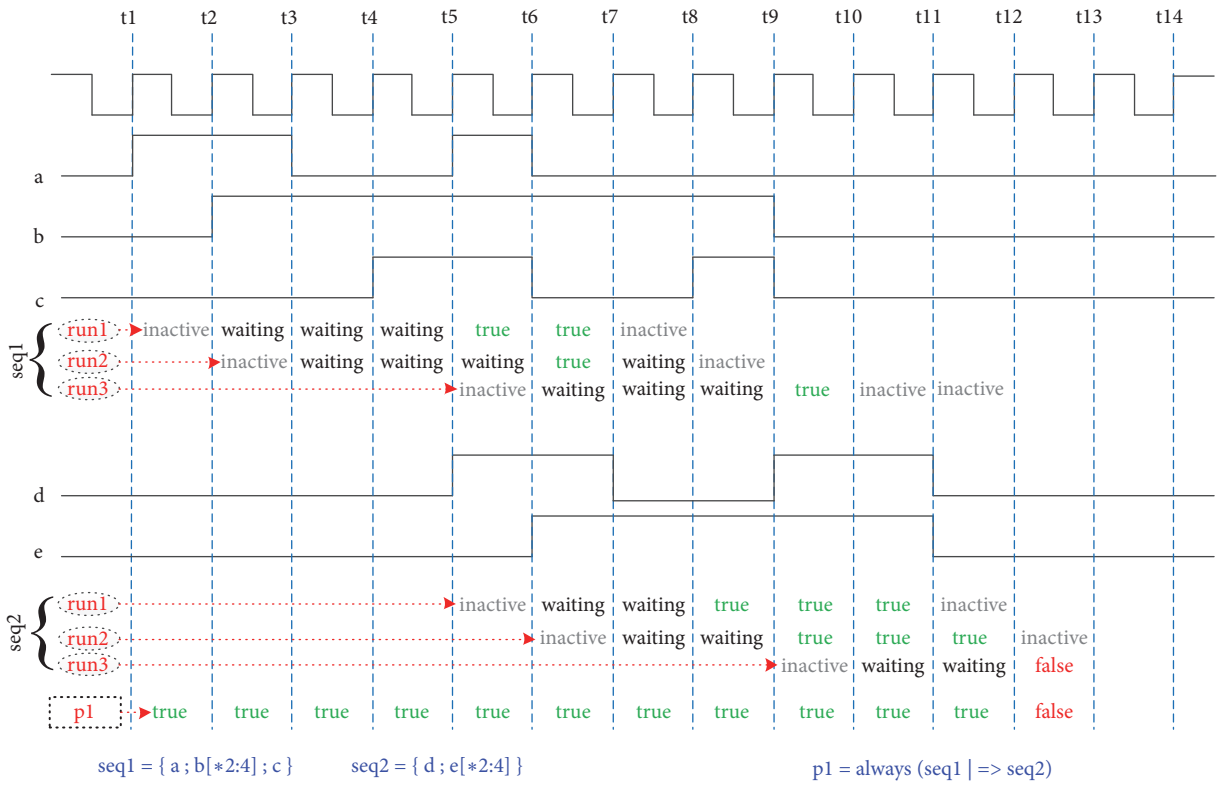


Figure 22: An example sequence evaluation.

```
1  -- Only one master at a time can have control
2  -- over the bus, so only one of the grant
3  -- signals can be High at a time.
4  pr1 g0 g1 rst=always
5         (
6           neg $ (sig g0) /\ (sig g1)
7         ) $ abort $ sig rst
8  -- If none of the masters is granted, any
9  -- request (not simultaneously) will lead
10 -- to a grant in the next clock cycle.
11 pr2 g0 g1 r1 rst = always
12       (
13         prev (neg((sig g0) \/ (sig g1)) /\ (sig r1))
14               --> (sig g1)
15        ) $ abort $ sig rst
```

Listing 4: Example properties for the AHB-based design.

of the sequence S2. The implication still holds if sequence S1 is not recognized in the signal trace. The sequence implication implemented in this work, also holds while S1 or S2 is pending (Waiting). Several ongoing recognitions of a sequence can have overlaps depending on the value of the input signals in the trace. Each evaluation of sequences S1 and S2 is called a *run* in this text. There are three overlapped runs for both sequences S1 and S2 shown as run1, run2, and run3 in Figure 22. The state of the evaluation of a sequence will be one of the followings at every time point:

(i) inactive: evaluation of the sequence is not started yet. This state is valid for both left and right sides of the suffix implication operator.

(ii) waiting: the sequence has been recognized in the trace so far but its evaluation is not finished yet. The

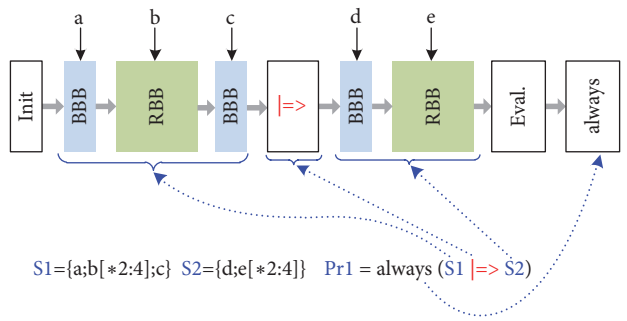


FIGURE 23: An example SERE structure in CλaSH.

final result of the evaluation depends on its input values in future clock cycles. This state is valid for both left and right sides of the suffix implication operator.

(iii) `true`: the sequence is recognized in the trace successfully (which can also be more than one clock cycle expressing several overlapped evaluation and recognition of it in the trace). This state is valid for both left and right sides of the suffix implication operator.

(iv) `false`: the evaluation of the sequence encountered a mismatch between the actual value in the trace and the expected value, which means the sequence is not recognized. This state is valid only on the right side of the suffix implication operator and implies a violation of the expected behaviour. However, if the left side does not hold the implication still holds.

A step by step evaluation of an example property which is composed of two sequences and a suffix implication operator `( |=> )` is shown in Figure 22. At the first positive-edge of the clock signal (which is marked with `t1`) all three input signals of sequence `S1` are `False` and the evaluation of the sequence is not initiated yet. Starting from time `t2`, with detecting a valid value on signal `a` , the evaluation enters into the `wait` state and holds in clock cycles `t5` and `t6` which expresses two consecutive recognition of sequence `S1` in the given trace. The input signal `a` is `True` in two consecutive clock cycles (`t2` and `t3`) which leads to two concurrent evaluation runs (`run1` and `run2`). The third evaluation run is created at time `t6` in parallel with the two ongoing evaluation runs. Any activation of the *antecedent* (`S1`) must be followed by at least one successful recognition of the *consequent* (`S2`) ; otherwise the evaluation will enter the `false` state. A corresponding evaluation run of `S2` for each `true` cycle of `S1` is initiated. This can lead to concurrent evaluation runs for the right side of the implication, as in this example it leads to three concurrent evaluation runs starting at `t6`, `t7`, and `t10`.

Both `run1` and `run2` reach to `true` states while `run3` ends in the `false` state which means that the property `p1` does not hold. The `false` state is entered because after the input signal `d` is `True` at time `t9`, at least two consecutive `True` values on signal `e` are expected, while it is `True` for only one clock cycle. Every sequence represented in CλaSH automatically sets up the corresponding synthesizable circuit for itself using function composition of Haskell and connecting the building blocks serially. Further details of the building blocks are explained later in this section. The general structure of the final circuit for the example sequence is shown in Figure 23. As shown in Figure 23, a sequence is assembled with a chain of building blocks connected serially. The sequence building blocks are divided into the following categories:

(i) `Basic building blocks (BBBs):` a BBB monitors the input signal from the DUV and the input signal from the previous stage and generates output signals which become the input set for the next stage.

(ii) `Repetition building blocks (RBBs):` an RBB, in contrast to a BBB, has a more complex functionality and structure because of its multicycle behaviour which leads to overlapped and concurrent evaluations.

(iii) `Other sequence building blocks:` this category includes all the building blocks except BBBs and RBBs, which are used for initialization, sequence evaluation, and some temporal operations (e.g., `init`, `always`, and `eval`) in the verification sequences.

The `always` building block checks the final evaluation result of the verification expression at every clock cycle. The `eval` building block is attached to the final stage of a sequence and feeds the next temporal operator building block (such as `always`) with the final evaluation result of the sequence. The suffix implication building block `( |=> )` activates the evaluation of its right side (consequent) in the next clock cycle if its left side (antecedent) holds in the current clock cycle.

Each building block has a different behaviour depending on which side of the suffix implication operator it is used. Building blocks on the left side of the suffix implication have less complex functionality than the same building blocks on the right side. In order to cover both cases, each building block has two working modes, *antecedent mode* and *consequent mode,* referring to its working modes on left and right side of the suffix implication operator, respectively. The microarchitecture of these building blocks in both antecedent and consequent modes is discussed in the rest of this section.

(1) Sequence building blocks in antecedent mode: each BBB simply monitors the value of an input signal and the value from the previous stage and generates the value for the next stage. The generated value will be `True` if both of its inputs are `True;` otherwise it will be `False`. The evaluation process takes only one clock cycle and there is no need to have overlapped evaluations. The equivalent circuit for a basic building block is shown in Figure 24.

The repetition building blocks implement the functionality of the repetition operators (e.g., `b[∗2:4]`) and in contrast to basic building blocks, they must handle overlapped evaluations that can happen because of their multicycle behaviour. The maximum number of overlapped evaluations is defined by Maximum Overlap Index (MOI) parameter.

Figure 25 shows the general architecture of the equivalent circuit for repetition operators. RBBs need more resources compared to BBBs and contain MOI number of counters to count the number of clock cycles the input signal (`DataIn`)

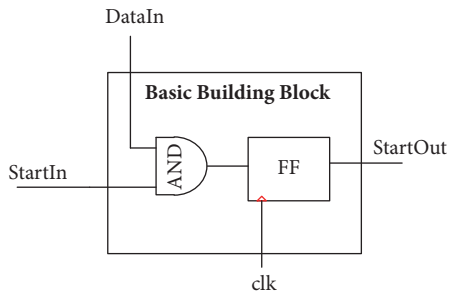


Figure 24: Structure of a basic building block.

is active. Each FSM runs an evaluation in case of overlapped evaluations in the RBB. Therefore, the maximum number of FSMs in an RBB is the same as the maximum number of overlapped evaluations which is `MOI`. Currently, MOI is set by the designer at design time and is not automated yet. For example, in case of `b[*2:4]`, MOI is set to 4 to ensure full evaluation of each counting process for signal `b` without missing any trace. Any `StartIn` signal initiates an FSM on the next clock cycle. If `StartIn` input becomes active again before the recently started FSM ends, another FSM will start in parallel to keep track of the new counting process and the new evaluation window (for example, at time points `t3` and `t6` in Figure 22).

If an FSM in an RBB detects the expected number of active values (or a range of repetitions as in the example of Figure 22) on the input signal, the output `StartOut(i)` will be activated in order to inform the next stage in the sequence. The outputs of all FSMs are merged into one output signal `StartOut` since the next stage does not need to discriminate between different evaluation processes running in the RBB block of the previous stage. The functionality of the FSMs may vary for different repetition operators. Figure 25(b) shows the behaviour of the FSMs for the repetition operator `[*N:M]` used in the example of Figure 22 as `b[*2:4]`. If an FSM is activated `(CntActive = true)`, the control moves to the state `LessThanN` in which it waits until the input signal is active for N consecutive cycles. Then the output is activated and the control moves to the LessThanM state and stays there until the input signal is active up to M consecutive clock cycles.

(2) Sequence building blocks in consequent mode: sequence building blocks are more complicated in consequent mode than antecedent mode considering the following feature of the suffix implication operator.

The implication holds if any recognition of the sequence on the left-side of the implication operator leads to at least one recognition of the sequence on the right side of it.

This feature implies that every single building block (BBB or RBB) in the right side of the implication must be aware of all overlapped evaluation runs going on at the current clock cycle, to check no evaluation ends without at least one recognition of the right side. This is done by monitoring each evaluation run in the building block individually. Handling overlapped evaluation runs in RBBs with a *range* of repetitions in input (which are called Ranged-RBBs) such as `a[*N:M]` is more complicated than BBBs and other RBBs. In these building blocks, a multicycle `StartOut` signal is needed to enable the next building block in the chain, while, in other building blocks, `StartOut` signal needs to be active only for one clock cycle which means the expected signal values are detected in the current clock cycle and enable the next stage in the chain. For example, the `{a[*4];b}` expression in a sequence holds if signal `a` is active for exactly four consecutive clock cycles starting from the time it is enabled by its previous stage, and `b` is active in the fifth clock cycle. When `a[*4]` holds, `StartOut` is activated for exactly one clock cycle which is used as the start signal (`StartIn` signal) for the next block in the sequence `(BBB(b))`. The next block in the sequence must hold in the clock cycle that it receives an active `StartIn` signal; otherwise the sequence is not recognized and the evaluation ends with a mismatch. The situation is different for a Ranged-RBB and the building blocks following it in a sequence. Ranged-RBBs, instead of a single cycle activation of the `StartIn` signal for the next building block, generate a multicycle window of active `StartIn` signal for the next building blocks in the sequence. This window is called *enable-window* in this text and it moves forward in the sequence at every clock cycle originating from the Ranged-RBB. In fact the enable-window is the number of clock cycles that the Ranged-RBB holds considering its input signal. All the building blocks following the Ranged-RBB in the sequence must hold *at least* once in the time-span of the moving enable-window; otherwise a mismatch will be reported. Each evaluation run generates its own enable-window for the next building blocks in the sequence which can lead to parallel active enable-windows in a Ranged-RBB. For example, if input signal `a` is continuously active for `K` clock cycles, the length of the enable-window for the expression `a[*N:M]` will be as follows in terms of clock cycles which is the number of clock cycles the expression holds:

$$EnableWindowLength = \begin{cases} 0, & \text{if } (K < N) \\ K - N, & \text{if } (N < K <= M) \\ M - N, & \text{if } (K > M) \end{cases} \quad (1)$$

Figure 26 shows an example property `(p2)` which includes a Ranged-RBB in the consequent part of the implication operator and its evaluation on an example signal trace. The antecedent part holds at time `t2`, which starts an evaluation run in the Ranged-RBB block `b[*2:4]` (shown as `run1`). At time `t3`, the second evaluation run (shown as `run2`) starts in parallel with `run1`. The expression `b[*2:4]` holds for three clock cycles from `t4` to `t7` which leads to an enable-window of length three in run1 in the Ranged-RBB operator. The next building block in the sequence `(BBB(c))` holds twice (at cycles `t5` and `t7`) in this enable-window and acknowledges the recognition of the sequence until this point. The enable-window is always passed to the next block in the sequence `(BBB(d))` from current block `(BBB(c))` with one clock cycle delay. In this case the shifted enable-window spans `t5` to `t8`. In this range of the enable-window, `BBB(d)` holds at `t6` and

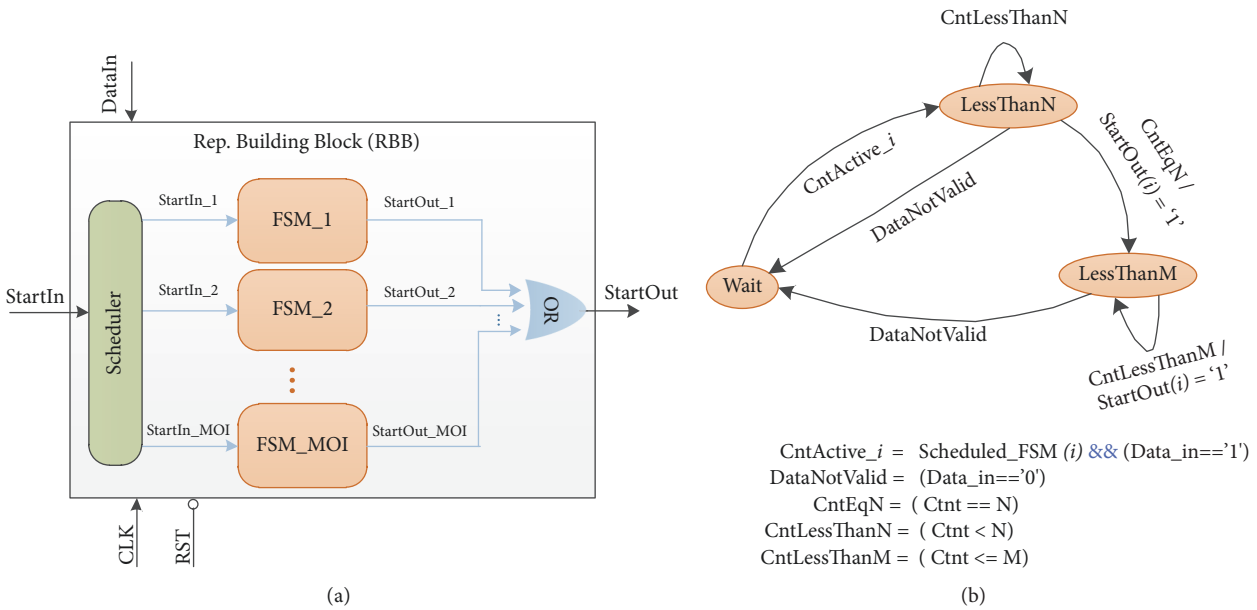


FIGURE 25: (a) Structure of an RBB in antecedent mode. (b) FSM of the [∗N:M] operator.

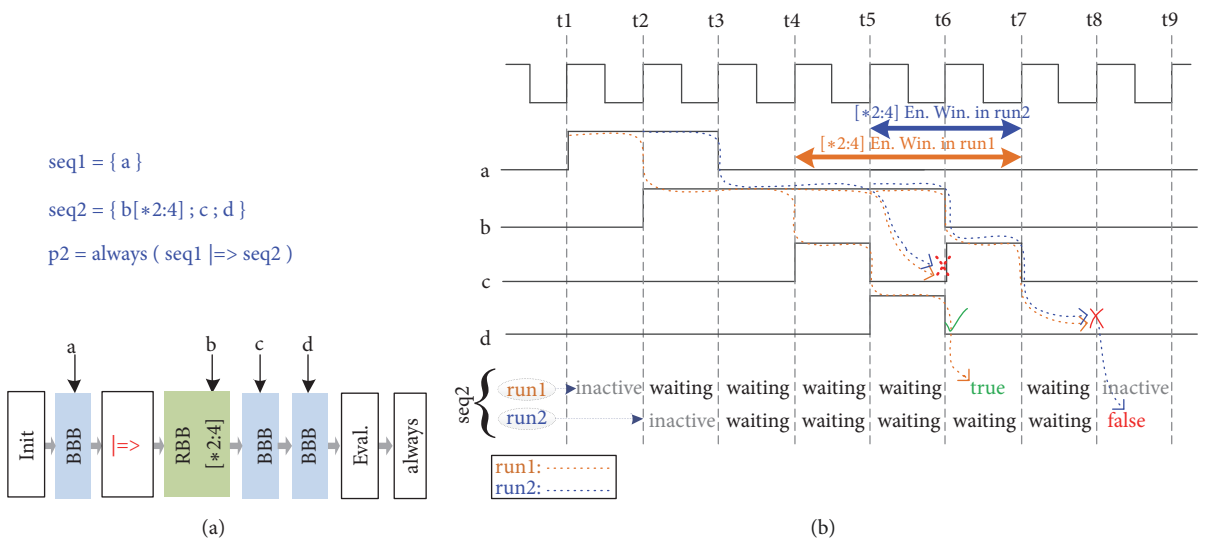


FIGURE 26: (a) An example property with sequences. (b) Overlapped *runs* and *enable-windows* in consequent mode.

completes a recognition of the sequence for `run1`. `run2` starts at `t3` in parallel with `run1` and generates an enable-window at `t5` which lasts for two clock cycles. An active value is detected for signal `c` in block `BBB(c)` in the enable-window, which acknowledges the detection of the sequence until this point. The enable-window is then passed to the `BBB(d)` block and spans `t6` to `t8` in which there is no activation of signal `d` and the sequence recognition process for `run2` fails which leads to a mismatch in recognizing the sequence in the provided signal trace.

In order to simplify the design of building block circuits, only one Ranged-RBB is allowed to appear in the consequent and antecedent parts of the suffix implication operator. Including more than one RBB in a sequence might need handling a more complex communication and is not addressed in this work. The detailed structure of a `BBB` in consequent mode is shown in Figure 27(a). `StartIn` and `StartOut` ports are used to enable the current block and the next block in the sequence, respectively. The `mismatch` signal is generated inside the building block and ORed with the input mismatch signal to reflect the global mismatch signal in the sequence. The input port `OverlapIn` is used to receive enable-windows from the preceding block in the sequence. This input is used by the FSMs to evaluate the overlapped evaluation runs in parallel. Maximum number of active FSMs at the same time is `MOI`. The `OverlapIn` is registered and passed to the next block in the sequence with one clock cycle delay.

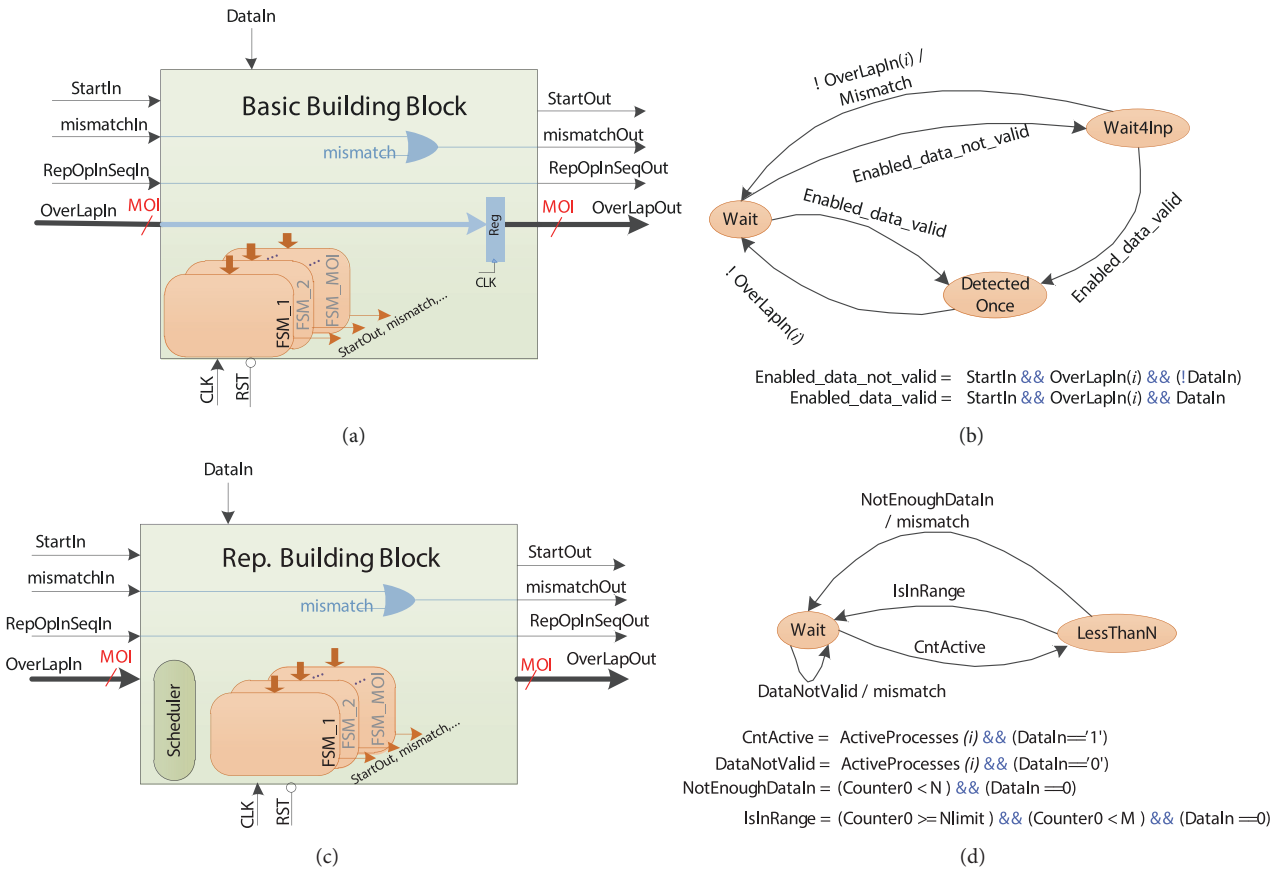


FIGURE 27: Building blocks in consequent mode.

The FSM shown in Figure 27(b) represents a simplified behaviour of one of the `MOI` identical processes running in a BBB in consequent mode. Each FSM waits for an enable-window on the corresponding `OverlapInp` port and after detecting an active enable-window it waits (in state `Wait4Inp`) until it captures a valid value for `DataIn` signal. If the enable-window ends without any detection of the active value of `DataIn` signal, the mismatch signal is activated and the recognition of the sequence fails.

Functionality of the BBB circuits is very simple (similar to Figure 24) if there is no RBB earlier in the sequence. The BBBs, following an RBB building block, need to consider the overlapped runs and the enable-windows of the preceding RBB which is more complicated than the single clock cycle monitoring of the input signals. This holds in both antecedent and consequent parts of the suffix implication. The input port `RepOpInSeqIn` is used to specify the working mode of the BBB. If this input is `true` it means that there is an RBB preceding the BBB and it should use the circuitry of Figure 27 to handle overlapped runs and enable-windows from the preceding building blocks. When this input is `false`, the simple circuitry similar to the one of Figure 24 can be used to handle the sequence recognition. This input is passed to the next building block in the sequence without any change. The detailed structure of a `Ranged-RBB` in consequent mode is shown in Figure 27(c). `StartIn` and `StartOut` ports are used to enable the current block and the next block in the sequence, respectively. The `mismatch` signal is generated inside the building block and ORed with the input mismatch signal to reflect the global mismatch signal in the sequence. Maximum `MOI` identical FSMs can be active at the same time to create overlapped runs and enable-windows. The output port `OverlapOut` is used to pass overlapped enable-windows to the next building block in the sequence. Port `RepOpInSeqOut` is a copy of the input port `RepOpInSeqIn`, to inform the next building blocks about the presence of an RBB in the sequence. The `Scheduler` process enables an FSM for each activation of the input signal `StartIn`. Any activation on `StartIn` port in a clock cycle means that the RBB must hold starting from that clock cycle. If there are already active FSMs, another FSM will be activated in parallel which creates an overlapped run and generates its corresponding enable-window on the `OverlapOut` port. A simplified FSM of the behaviour of a `Ranged-RBB` is shown in Figure 27(d). Each FSM waits in the wait state until it is activated by the scheduler. The `DataIn` input must continuously be `true` at least `N` clock cycles before it becomes `false` (in state `LessThanN`); otherwise, it will fail to recognize the sequence. In order to ease the automatic assembling of the sequences by chaining the building blocks, as shown in Figure 26(a), BBBs and RBBs have the same interfaces and ports. `OverlapIn` input port is not used in RBBs but it exists

Table 1: Implemented operators.

PSL Operator	In Haskell	PSL Operator	In Haskell
and	/\	a[∗N]	a | = | N
or	\/	a[=N:M]	a | ~= | (N,M)
not	neg	before	before
− >	− − >	until_	until_
| =>	==>	eventually!	eventually
rose	rose	until	until
prev	prev	always	always
prev(a,N)	prev(a,N)	fell	fell
a[+]	repPlus(a)	a[∗N:M]	a | ~~ | (N,M)
before_	before_	;	...

because of the interface-compatibility. Boolean and temporal operators supported in this work are shown in Table 1. The BBB operator, which is not shown in Table 1, is represented by `se` in the CλaSH descriptions of the sequences. The `...` operator as shown in Table 1, is equivalent to semicolon in SEREs which is used for representing the clock edge between sequence elements. Since the output values of the sequence operators supported in this work are registered with the clock, the `...` operator only needs to combine its input operands serially. Its description in CλaSH is as follows:

```
(...) a b = b.a.
```

As an example, consider the following property in PSL:

```
always {i1[+]} |=> {i2;i3;i4[*3:6]}.
```

This property is described in CλaSH as follows:

```
always (((repPlus i1)) ==>
((se i2)...(se i3)...(i4 |~~| (3,6)))).
```

The assertion checkers generated in this work are not optimized in resource usage or operational frequency. As an example the circuit for the expression `{a;b}|=> {c[* 0:1];d}` uses 4 flip-flops and 3 look-up tables on the target FPGA according to [5] while the equivalent sequence using our approach uses 49 flip-flops and 70 look-up tables.

5.1. High-Level Properties in CλaSH. Using Haskell as the unified language for describing both the DUV and its synthesizable properties offers an easier integration of properties into the DUV and allows the designer to use parametrization, higher-order functions, polymorphism, and other high-level structures and functions of Haskell for describing design properties. As an example, a general and parametric property is described on a 64-bit block of data in Figure 28 which can be specialized to variety of different properties with parameters f2 and f1 which are Haskell functions themselves. The `map` function in the expression, is a built-in function in CλaSH that applies a function `(f1)` on an array of inputs `(i)`. For example, in order to specialize the template to a property that checks if the value of all input bytes `(i)` are less than 128, the designer can simply pass the following parameters to the template property:

```
lT128 = always (p_template (lT 128) aND High i)
```

in which, `(lT 128)` (passed as `f1`) is a function that checks if the input byte is less than 128, and `aND` (passed as `f2`) is the logical *AND* function which is folded from left to right

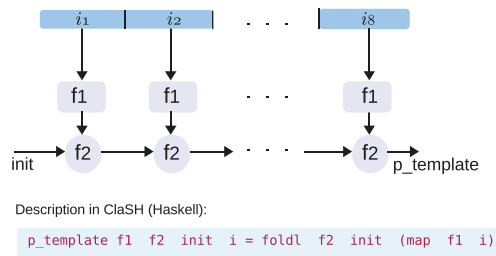


Figure 28: A parametric property in Haskell.

with `init` as its initial value. Another property that checks if the input bytes `(i)` are nonzero `(nEq 0)` is described using the same property template as follows:

```
always(p_template (nEQ 0) aND High i).
```

6. RTR Implementation of Assertion Checkers

As mentioned in Section 1, placing all the assertion checkers of the design in one verification module can lead to a large, resource demanding module, and the amount of the available resources on the target implementation platform might not be enough to implement the verification module. A solution for this problem is to partition the verification module into smaller parts and use partially reconfigurable platforms to implement and run them on the hardware in a time-multiplexed manner. By using this method, each partition will include only a subset of assertions and can be deployed into the partially reconfigurable hardware on its time-slot, to check for a subset of the design properties. Current RTR designs have one or more RRs defined in the target FPGA and can be implemented using the partial reconfiguration flow from Xilinx. In this work, we use the RRs on the target FPGA for implementing assertion-checker circuits, in addition to their use for hosting different reconfigurable modules (RMs), by integrating assertion checkers with the RTR design descriptions in high levels of abstraction discussed in Section 3. The verification circuits can be programmed into the desired RR by the reconfiguration management software, on demand.

In order to map verification-circuit partitions into each RR, the limitations of the RR in terms of the number of inputs/outputs and the amount of hardware resources

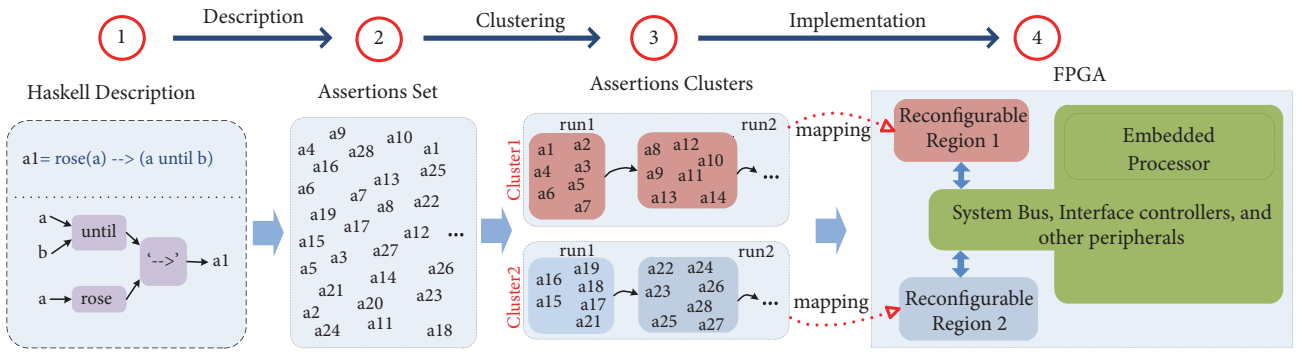


FIGURE 29: The proposed verification flow.

available in the RR, should be considered. Each subset must physically fit into the desired RR considering these limitations. As a result, instead of checking the design with the whole set of assertion checkers at once, the assertion subsets can be programmed one by one in the RR until all of them have been covered, in a time-multiplexed manner. An RR can host any reconfigurable module if resources required by the module (including input/output ports) are available in the RR. Since the routing of the static part is fixed, the operational frequency of the static part is not affected by configuring different modules into the RR.

Usually assertion circuits are made of simple logical operations, and the number of input signals used in the circuit of the whole assertion set grows faster than the number of hardware resources needed. Considering the fact that an RR needs special communication structures such as bus-macros or any kind of proxy logic to communicate with the static part of the design, and these structures are implemented with logic (such as look-up tables) themselves, increasing the number of inputs/outputs of the RR and changing the design just for the verification purpose are not an efficient solution.

We propose a partitioning algorithm that considers the limitation on the number of inputs/outputs of the *existing* RRs and the amount of available resources in them and try to minimize the reconfiguration overhead (i.e., reduce the number of reconfigurations). The reconfiguration overhead of this approach is directly proportional to the total number of assertion partitions, or, in other words, the number of reconfiguration candidates for RRs, for verification purposes. Lowering the total number of partitions lowers the number of reconfigurations in the target FPGA. The problem of achieving the minimum number of subsets can be mapped into the so-called *bin-packing* problem, which is proven to be NP-hard [28]. In bin-packing problem, objects of different volumes must be packed into a finite number of bins each with a specific size, in a way that minimizes the number of bins used. In order to achieve close-to-optimal subsets we use a two-phase partitioning approach shown in Figure 29 which depicts the main steps of the verification methodology. It starts with describing the DUV and its assertions in CλaSH. The described properties are extracted into a set which is then clustered into *assertion clusters* using the first phase of the partitioning algorithm. Assertion checkers in each cluster are intended to be mapped into an RR on the reconfigurable platform. If the hardware resources required to implement a cluster of assertion checkers are more than the available resources on the corresponding RR, a second phase of the partitioning algorithm is used to partition it into two or more *reconfiguration runs*, until each assertion-subset fits into the target RR. Each reconfiguration-run will be mapped into the corresponding RR with commands issued by the rest of the system (e.g., software) based on the desired scheduling. The general architecture of the verification circuitry is shown in Figure 30. Each RR is fed by either *normal-mode* or *verification mode* inputs. The normal-mode inputs come from the reconfiguration candidates in regular operational mode of the design, while the verification mode inputs are the input signals of the assertion-checker subsets, selected during the verification mode. The `AssertInpSet_i` input in Figure 30 represents the input set for assertion-cluster (i). The corresponding input set is selected when the RR is programmed to function for a specific assertion-cluster by the verification control process. The verification control process is a software process that runs on the embedded processor of the target FPGA and can program different reconfiguration runs into the RR using the corresponding partial bitstreams stored in the system memory, as well as the control signals of the multiplexers for input-sets of the clusters. The reconfiguration overhead of this approach in terms of time depends on the total number of reconfigurations for verification purpose which can be calculated as follows:

$$\texttt{NumOfReconfigurations}=\sum_{i=1}^{N_{cls}} N_{runs}(i)$$

in which N_{cls}: *Number of clusters after partitioning the assertion set and* $N_{runs}(i)$: *Number of runs in cluster* `i`.

In order to minimize the hardware overhead, which is the multiplexers at input of the RR, as well as the reconfiguration-time overhead, which depends on the total number of reconfiguration runs, the value of `NumOfReconfiguration` should be minimized. Since this can be mapped into a variation of the *bin-packing* problem (by considering the limitations on the inputs/outputs) which is an NP-hard problem, a heuristic algorithm is used for partitioning the assertion set which tries to get close to the minimum value of `NumOfReconfiguration` by first minimizing the number of clusters (N_{cls}) and then minimizing the number of runs for each cluster ($N_{runs}(i)$).

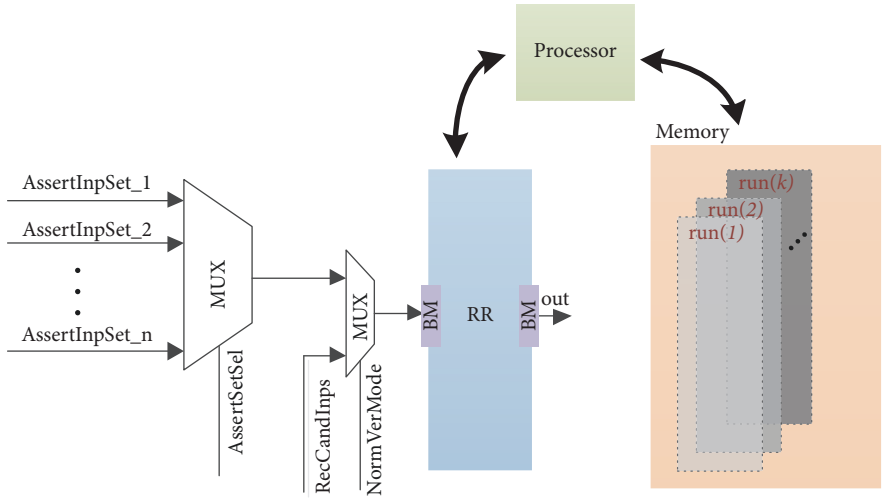
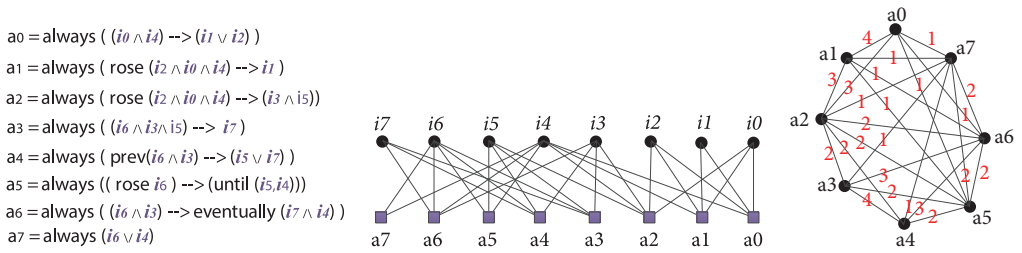


FIGURE 30: An RTR architecture for verification.



(a) Assertions described in Haskell (b) Input Dependency Graph (IDG) (c) Assertion Dependency Graph (ADG)

FIGURE 31: Dependency graphs for the example assertions.

The clustering algorithm uses specific graph-based structures to represent assertions and their dependencies. The definitions of the these graphs are as follows.

Definition 1. **Input dependency graph (IDG)** is an undirected graph which is described as follows:

$IDG = (V, E)$.

$V = \{v_p\} = I \cup A$, the set of all nodes in the graph.

$E = \{e_{pq} \mid p \in I, q \in A, p \in inps(q)\}$, the set of edges between A and I.

$A = \{a_p\}$, the set including all assertions.

$I = \{i_p\}$, the set including all input signals of assertions.

$inps(a_p)$: set of input signals for assertion a_p.

N : total number of assertions.

An IDG of an assertion set shows how the input signals are distributed between assertions. For example, the IDG of the assertion set of Figure 31(a) is shown in Figure 31(b).

Definition 2. **Assertion dependency graph (ADG)** is a weighted undirected graph which is described as follows:

$ADG = (V, E, W)$.

$V = A$, the set of all nodes in the graph.

$E = \{e_{pq} \mid inps(p) \cap inps(q) \neq \emptyset, p, q \in A\}$, the set of edges between assertions with common inputs.

$W = \{w_{pq} \mid w_{nm} = \|inps(n) \cap inps(m)\|, n, m \in A\}$, the weight of the edge between p and q.

$A = \{a_p\}$, the set including all assertions.

$inps(a_p)$: the set of input signals for assertion a_p.

An ADG represents the dependency for each pair of assertions in terms of their common input signals. Assertion-pairs with high number of common inputs are considered *strongly* dependent in ADG, while the pairs with low number of common input signals are considered *loosely* dependent. The corresponding ADG for the assertions of Figure 31(a) is depicted in Figure 31(c).

The heuristic clustering algorithm, which uses these graphs is shown in Algorithm 1. The main goal in this algorithm is to minimize the number of clusters by making clusters as large as possible, considering the limitation on the number of inputs/outputs of the target RR. In order to reach that goal, the two most strongly dependent assertion-sets will be merged while the number of inputs for the merged assertion set respects the limitation on the number of inputs for the target RR.

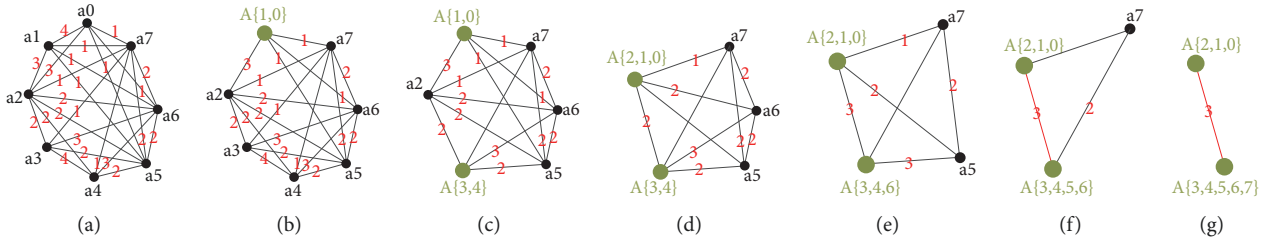


Figure 32: An example run of the proposed partitioning algorithm.

input: *IDG* and *ADG* of the assertion-set
output: Clusters for RRs
1 *// The main loop for creating clusters*
2 $ADG' = ADG$;
3 **while** (!*EdgesAllMarked*) **do**
4 *// Find a non-marked Edge with maximum weight*
5 e_{nm} = `FindMaxW` (ADG');
6 **if** ($|inps(a_n)| + |inps(a_m)| - |inps(a_n) \cap inps(a_m)| \leq |RR_{inps}|$ **then**
7 *//Merge the strongly-connected pair*
8 `MergeNodes` (a_n, a_m, *IDG*);
9 *// Update the weights in theADG'*
10 `UpdateW` (ADG');
11 **end**
12 **else**
13 `MarkEdge` (e_{nm});
14 **end**
15 **end**
16 *// A Best-Fit bin-packing on still-mergable clusters*
17 `BFPacking` (ADG');
18 *// ADG' contains the final set of clusters*
19 `WriteOutput` (ADG');

Algorithm 1: The proposed clustering algorithm.

The inputs for the partitioning algorithm are the IDG and ADG of the assertion set and the output of the algorithm is an undirected graph in which each node represents a cluster and each edge represents the degree of dependency between the two assertion-cluster end nodes. In this algorithm, the strongly connected nodes are continuously merged to maximize the size of the clusters while the total number of inputs of the growing cluster is lower than or equal to the maximum number of inputs allowed in the targeted RR. A greedy approach is used to merge clusters and the most dependent pair of nodes are merged first if the restriction for the number of inputs holds on the merged cluster. If the restriction is not satisfied on the merged cluster, the merging is undone and the edge in between is marked while continuing with the next most dependent pair. The greedy merging of connected pairs continues until all edges in the resulting graph are marked and further merging of the connected clusters is not feasible because of the restriction on the number of inputs for each cluster. Finally, a variant of the *best-fit* bin-packing algorithm is applied by merging the not-connected and mergeable assertion clusters on the resulting clusters to make compact clusters without violating the restriction on the number of inputs. The best-fit algorithm is one of the several heuristics to solve the bin-packing problem. In the best-fit approach, during choosing an object for the target bins, an effort is made to waste as less bin-space as possible. Further details on the best-fit algorithm and the bin-packing problem can be found in [28].

The next phase of the partitioning algorithm is for creating reconfiguration runs for each cluster in order to minimize its number of runs. The best-fit heuristic is used here as well to partition each assertion-cluster into reconfiguration runs constrained by the amount of resources available in the target RR and the amount of resources needed for the cluster. The capacity of the RR is considered as the capacity of the *bin* and assertions of a cluster are considered as items to be packed in that bin. An example execution of the partitioning algorithm on the assertion set of Figure 31(a) is shown in Figure 32 for an RR with 6 inputs. In Figure 32(a) the ADG for the assertion set is shown which is followed by pair-merging steps from `(b)` to `(g)`. In `(b)`, assertions `a1` and `a0` are merged into assertion set `A{1,0}`. The greedy merging is continued until `(f)` in which assertion-sets `A{2,1,0}` and `A{3,4,5,6}` can not be merged because of the restriction on the number of the

```
a0  = always ( ((sig i0) /\ (sig i1)) --> (before ((sig i2),(sig i4))) )
a1  = ((se i9) ... (se i13) ... (se i6))  ==> (i4 |~~| (2,4))
a2  = ((repPlus i3) ... (se i15)) ==> ((se i13) ... (se i12) ... (se i0))
a3  = ((repPlus i8) ... (se i11) ... (se i13)) ==> ( (repPlus i3) ... (se i9))
a4  = always ( (sig i0) --> ( (sig i2) \/ (sig i3) \/ (sig i4) \/ (sig i5) ))
a5  = ((se i8) ... (se i9) ... (se i11)) ==> (i3 |~~| (3,5))
a6  = always (((sig i3)  /\ (sig i5) /\ (sig i6)) --> (rose(i7)))
a7  = always ( (rose(sig i14))  -->   (before ((sig i5),(sig i6))) )
a8  = always( ( (sig i12) /\ (sig i11) /\ (sig i13)) --> ( (sig i15) /\ (sig i0)) )
a9  = always ( ((rose (sig i3)) /\ (rose (sig i4))) --> ((sig i6) \/ (neg (sig i7))) )
a10 = always ((sig i11) \/ (sig i13))
a11 =  ((se i13) ... (i3 |=| 5)) ==> (i8 |~=| (4,6))
a12 = ((se i7) ... (se i4)) ==> ((i0 |=| 2) ... (se i3) ... (se i1))
a13 =  ((se i7) ... (se i0)) ==> ((se i6) ... (se i2) ... (i5 |~=| (3,6)) )
a14 =  ((se i7) ... (se i6) ... (repPlus i4)) ==> ((se i1) ... (se i2) ...(i5 |~=| (3,6)))
a15 = ( (se i10) ... (i14 |~~| (4,8)) ) ==> ( (se i9) ... (i6 |~~| (3,7)) ... (se i13) )
a16 =  ((se i0) ... (se i0) ... (repPlus i4)) ==> ( (se i3) ... (se i3) ... (i5 |~=| (3,6)) )
a17 = ( (i13 |~=| (3,8)) ... (se i14) ) ==> (i10 |~~| (3,9))
a18 = ( (se i6) ... (i9 |=| (3)) ) ==>  (i10 |~~| (3,9))
```

Figure 33: An example assertion set.

inputs and the edge between them is marked (in red). Finally the assertion `a7` is merged with the set `A{3,4,5,6}` which leads to two clusters for the input assertion set targeted for the RR with 6 inputs.

To experiment the assertion description in CλaSH, assertion circuit generation, and the partitioning algorithm, the set of assertion checkers shown in Figure 33 is used, which consists of 19 assertion checkers. The synthesis result on a Zynq-7000 FPGA is listed in Table 2. Considering the high operational clock-frequencies of the checkers compared to typical frequency of complex designs on FPGAs, they are unlikely to be on the critical-path to limit the performance of the whole design. Figure 34 shows the results of running the partitioning algorithm on the ADG and IDG of this example assertion-checker set. As shown in Table 3, three clusters or partitions are created for an RR with 8 inputs, after running the partitioning algorithm.

7. The Unified Design and Verification Flow

The complete design and verification flow used in this work is shown in Figure 35. The steps shown in green in the flow represent the main contributions of this work while other steps (shown in blue) represent the existing design steps in the conventional design flows. The flow starts with a high-level description of the design and its assertions using CλaSH. The RTR design together with its assertions are described by exploiting the proposed RTR structures and assertion-operators existing in the libraries. The Haskell description is then simulated and translated into VHDL using the CλaSH tool. An initial synthesis is done by Xilinx ISE/Vivado synthesis tool to determine the amount of hardware resources needed for each assertion. This stage can easily be adapted to other resource estimation or synthesis tools as long as implementing RTR designs on partially reconfigurable hardware is supported in the tool-chain. The results from the resource estimation stage are used as input data for the partitioning stage. Based on size, number of inputs, and IDG/ADG graphs of the assertions, the assertion set is partitioned into clusters and reconfiguration runs as discussed in the previous section. After the partitioning stage, a synthesizable module is generated for each reconfiguration-run of each RR. The reconfiguration runs are used as the reconfiguration candidates for the RR in the Xilinx's partial reconfiguration flow. We use Xilinx's PR flow in Vivado 2015.2 for implementing RTR designs on FPGAs with DPR support such as Zynq, as discussed in Section 3.

8. Conclusion and Future Work

With the increasing number of applications for reconfigurable platforms such as FPGAs supporting DPR, the need for high-level design languages and tools increases to cope with the productivity gap. Some of the FPGAs from Xilinx

Table 2: Synthesis results on Zynq-7000 FPGA.

Name	FFs	LUTs	Max Frequency (MHz)
$a0$	4	15	833
$a1$	35	108	455
$a2$	19	24	733
$a3$	43	80	570
$a4$	1	3	1221
$a5$	35	125	450
$a6$	3	4	940
$a7$	5	11	841
$a8$	1	3	1221
$a9$	3	4	1131
$a10$	1	2	1221
$a11$	72	168	450
$a12$	60	217	470
$a13$	50	111	460
$a14$	62	125	501
$a15$	80	240	431
$a16$	62	129	501
$a17$	81	225	423
$a18$	61	170	423

Table 3: Partitioning results.

Cluster number	Checkers in the cluster	FFs	LUTs	Max. freq.
1	{a2, a3, a5, a8, a10, a11}	171	402	450
2	{a1, a7, a15, a17, a18}	262	754	423
3	{a0, a4, a6, a9, a12, a13, a14, a16}	245	608	460

are among the common platforms for implementing RTR designs. However, the DPR flows from Xilinx still require the designer to have low-level knowledge on DPR. Dealing with low-level and time-consuming steps in the DPR design flow can limit the design space exploration. Design tools and description languages with higher levels of abstraction are required to improve the design space exploration and decrease the productivity gap for the designer. A flow for high-level modelling and implementation of RTR designs as well as their verification, using a unified design language, is presented in this article. A Haskell-based functional HDL called CλaSH is used to describe RTR designs and verification expressions.

A design flow including modelling of RTR designs in high levels of abstraction using the functional programming language Haskell, as well as its implementation on partially reconfigurable FPGAs from Xilinx, is proposed in this article. The proposed approach uses features such as higher-order functions, strong function composition, and parametrization in Haskell to model RTR components and reconfiguration candidates. A Python script called rclash, which uses the CλaSH tool as its Haskell-to-VHDL translation core, is used for generating a simulation-only model of the design in VHDL, integrating the PR design flow from Xilinx, and generating a reconfiguration management software in C. An example design including an RTR coprocessor is implemented on a Zynq FPGA from Xilinx using the proposed approach. Adding support for more FPGAs and their development boards and generating a more precise simulation model of the RTR design from Haskell descriptions are the potential future development tracks.

The proposed verification approach in this work is ABV, with assertion checkers implemented on the reconfigurable hardware. The design properties and their assertion checkers are described in the same language as the DUV and simulated with the CλaSH tool together with the DUV. The design properties and assertion checkers described in Haskell language are translated into synthesizable VHDL with the CλaSH tool together with the rest of the design. A modular approach is used for describing the design properties. The design properties are composed of the proposed Boolean and temporal operator building blocks which are implemented as FSMs and combined by using Haskell's strong function composition features.

A complete flow is proposed by combining the proposed RTR design flow and the proposed ABV approach, in which Haskell (CλaSH) is used as the unified language for describing the design and its properties. The set of assertion checkers

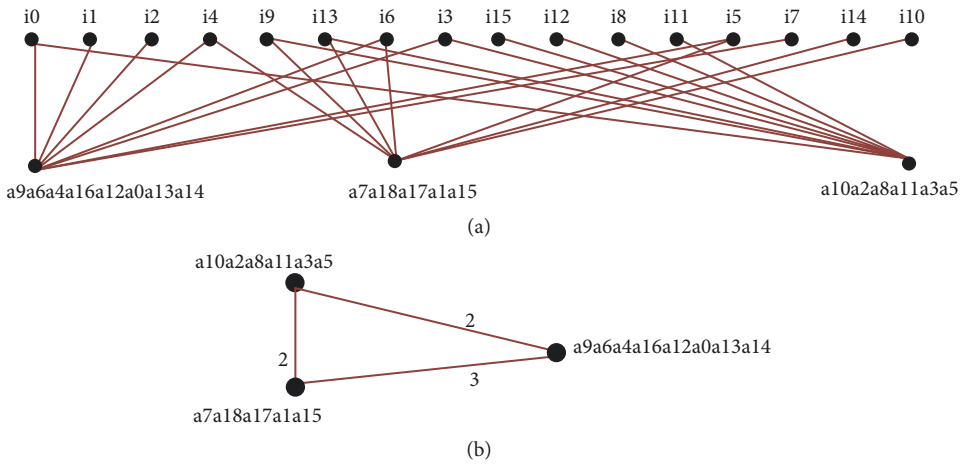


Figure 34: (a) IDG of the partitioned checker set. (b) ADG of the partitioned checker set.

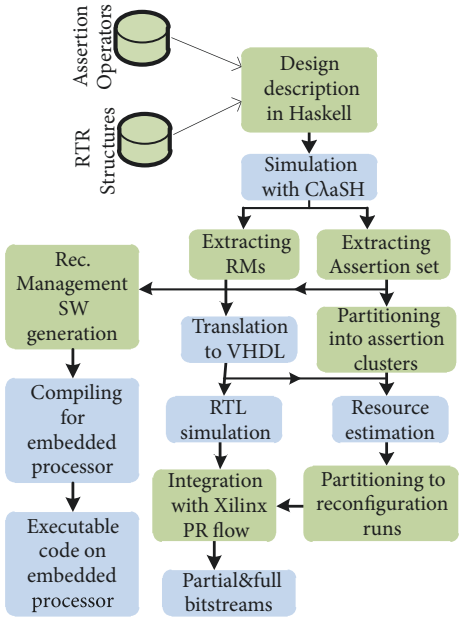


Figure 35: The complete design and verification flow.

is partitioned into smaller subsets with a proposed partitioning algorithm to fit the reconfigurable region on the target FPGA. Currently, a limited subset of LTL and sequence operators is supported in this work, and adding more operators to the set of supported operators can be a potential future work. The proposed assertion-checker circuits are not optimized for performance or resource usage. Improving their performance and resource usage by revising their implementation method is another possible future work.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors would like to thank Christiaan Baaij and Professor Jan Kuper from the Faculty of Electrical Engineering, Mathematics and Computer Science of the University of Twente, for providing the CλaSH tool and their valuable support for it.

References

[1] C. P. R. Baaij, *Digital circuit in CλaSH: functional specifications and type-directed synthesis*, University of Twente, Enschede, 2015.

[2] http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf , 2015.

[3] D. Borrione, M. Liu, P. Ostier, and L. Fesquet, "PSL-Based Online Monitoring of Digital Systems," in *Applications of Specification and Design Languages for SoCs*, A. Vachoux, Ed., Springer, Dordrecht, 2006.

[4] K. Morin-Allory and D. Borrione, "Online Monitoring of Properties Built on Regular Expressions Sequences," in *Advances in Design and Specification Languages for Embedded Systems*, S. A. Huss, Ed., Springer Netherlands, Dordrecht, 2007.

[5] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers*, Springer Netherlands, Dordrecht, 2008.

[6] F. Eibensteiner, R. Findenig, and M. Pfaff, "SynPSL: Behavioral Synthesis of PSL Assertions," in *Computer Aided Systems Theory - EUROCAST 2009*, vol. 5717 of *Lecture Notes in Computer Science*, pp. 69–74, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[7] M. Gao, H.-M. Chang, P. Lisherness, and K.-T. Cheng, "Time-multiplexed online checking," *Institute of Electrical and Electronics Engineers. Transactions on Computers*, vol. 60, no. 9, pp. 1300–1312, 2011.

[8] M. Gao and K.-T. Cheng, "A case study of time-multiplexed assertion checking for post-silicon debugging," in *Proceedings of the 2010 15th IEEE International High Level Design Validation and Test Workshop, HLDVT'10*, pp. 90–96, usa, June 2010.

[9] R. Andreas, *Describing and simulating dynamic reconfiguration in SystemC exemplified by a dedicated 3D collision detection hardware [Ph.D. thesis]*, University of Bonn, 2008.

[10] A. Siddiqi and O. L. De Weck, "Modeling methods and conceptual design principles for reconfigurable systems," *Journal of Mechanical Design*, vol. 130, no. 10, pp. 1011021–10110215, 2008.

[11] Simon. Thompson, Simon Thompson Haskell: The Craft of Functional Programming. Addison Wesley, 1996.

[12] IEEE Std 18502005, IEEE standard for property specification language. Pages 1143, 2005.

[13] A. Pelkonen, K. Masselos, and M. Cupak, "System-level modeling of dynamically reconfigurable hardware with SystemC," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, IEEE, p. 8, Nice, France.

[14] J. Vidal, F. De Lamotte, G. Gogniat, J.-P. Diguet, and S. Guillet, "Dynamic applications on reconfigurable systems: From UML model design to FPGAs implementation," in *Proceedings of the 14th Design, Automation and Test in Europe Conference and Exhibition, DATE 2011*, pp. 1208–1211, fra, March 2011.

[15] J. Vidal, F. De Lamotte, G. Gogniat, J.-P. Diguet, and P. Soulard, "UML design for dynamically reconfigurable multiprocessor embedded systems," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, DATE 2010*, pp. 1195–1200, deu, March 2010.

[16] M. Streub et al., "System-level modeling and performance simulation for dynamic reconfigurable computing systems in SystemC," in *proceeding of In Methoden und Beschreibungssprachen zur Modellierung und Verikation von Schaltungen und Systemen*, 59, 68 pages, 2007.

[17] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-Level Modeling of Dynamically Reconfigurable Co-processors," in *Proceedings of the 14th International Conference on Field Programmable Logic*, Lecture Notes in Computer Science, pp. 881–885, Springer Berlin Heidelberg, 2004.

[18] K. Asano, J. Kitamichi, and K. Kuroda, "Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems," *Journal of Computers*, vol. 3, no. 2, pp. 55–63, 2008.

[19] I. R. Quadri, S. Meftali, and J.-L. Dekeyser, "MARTE based modeling approach for partial dynamic reconfigurable FPGAs," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia 2008*, pp. 47–52, 2004.

[20] F. Dittmann and A. Rettberg, "Design of partially reconfigurable systems: from abstract modeling to practical realization," in *Proceedings of the In proceedings of the 1st International Workshop on Reconfigurable Computing Education*, 2006.

[21] C. Tseng and P. Hsiung, "UML-Based Design Flow and Partitioning Methodology for Dynamically Reconfigurable Computing Systems," in *Embedded and Ubiquitous Computing – EUC 2005*, vol. 3824 of *Lecture Notes in Computer Science*, pp. 479–488, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[22] C.-H. Huang and P.-A. Hsiung, "UML-based hardware/software co-design platform for dynamically partially reconfigurable network security systems," in *Proceedings of the 13th IEEE Asia-Pacific Computer Systems Architecture Conference, ACSAC 2008*, twn, August 2008.

[23] C. Baaij, "CλaSH tutorial," https://hackage.haskell.org/package/clash-prelude-0.10.7/docs/CLaSH-Tutorial.html, 2014.

[24] B. N. Uchevler, "The RCLaSH project," https://bitbucket.org/uchevler/rclash.

[25] A. Habibi, A. Gawanmeh, and S. Tahar, "Assertion based verification of PSL for systemC designs," in *Proceedings of the 2004 International Symposium on System-on-Chip*, pp. 177–180, fin, November 2004.

[26] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 406857, 17 pages, 2011.

[27] B. N. Uchevler and K. Svarstad, "Synthesizable assertion checkers in high levels of abstraction," in *Proceedings of the 2013 IEEE 20th International Conference on Electronics, Circuits, and Systems, ICECS 2013*, pp. 859–864, are, December 2013.

[28] S. S. Skiena, *The Algorithm Design Manual*, Springer-Verlag, London, UK, 2008.